SIO2 SDK v2.x
GETTING STARTED MANUAL
FOR
XCode, Eclipse and VC++

(last revision 2011/06/23)

1. Introduction

Welcome to this getting started tutorial for SIO2 Engine v2.x. In this document you will first get an overview of the SIO2 Engine SDK, and learn what is required in order for you to get you started right away.

Then you will learn how to build a simple casual game from scratch that will allow you to touch base with all the basics such as physic, lighting, instancing, how to handle materials and textures as well as game logic and game states.

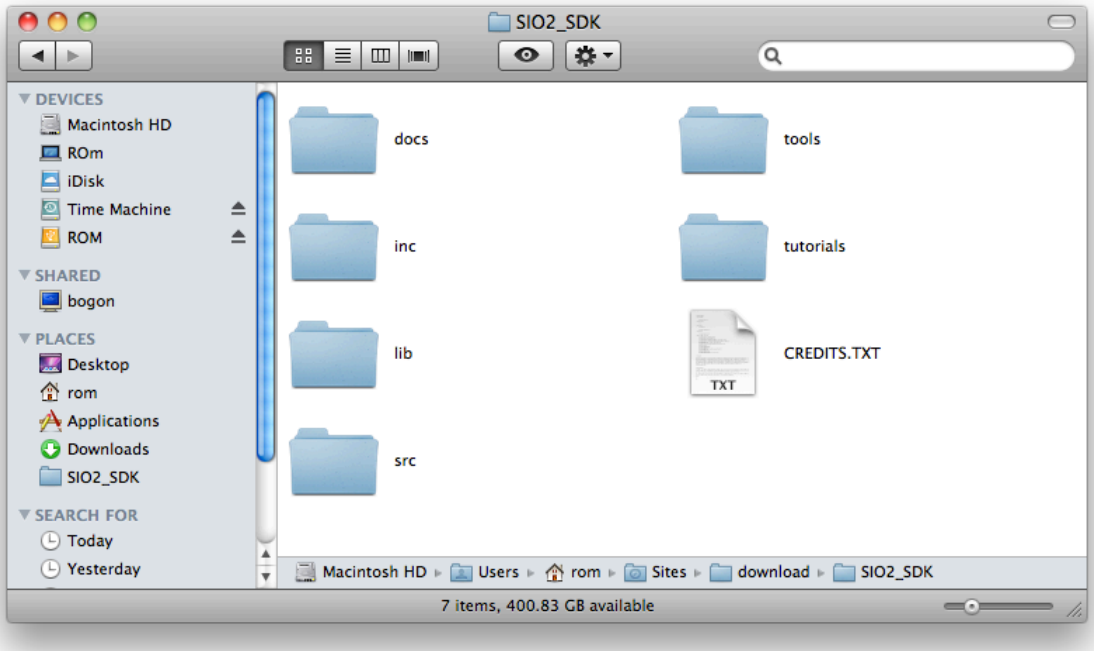
SIO2 SDK Overview

In order to get you started, you first download the the latest version of the SIO2 Engine SDK, available at the following address: http://sio2interactive.com/download.php.

Once you've downloaded the SIO2 SDK, simply un-compress the ZIP file in a directory that you

have read and write access (no other installation process is required).

The SDK file architecture should look like this:



- `SIO2_SDK/docs`: Contain an HTML file that will automatically redirect you to http://api.sio2interactive.com, where you can find the latest SIO2 API documentation, as well as the SIO2 Exporter manual for Blender, Maya and 3D Studio Max (available at the top of the left side treeview).

- `SIO2_SDK/inc`: This directory contain all the header files required by SIO2 as well as the other external libraries embedded inside the SIO2 Engine core.

- `SIO2_SDK/lib`: Contains pre-compiled static libraries (for iOS, MacOS, Windows and Android) used by SIO2 internally, along with their respective license(s), as well as their project file in order for you to be able to re-compile them from scratch for your targeted platform if needed. In addition for Windows developers, this directory contains all the necessary dependencies to mimic an iPhone, iPod Touch and iPad game development environment right on your Windows box, theses dependencies are required in order to be able to run SIO2 for your specific version of Windows. In addition by default all the dependencies are by default available for 32bits system, if you are using a 64bits system please make sure that you download the appropriate version for your system.

- `SIO2_SDK/src`: In this folder you can find all the source files used by the external SIO2 dependencies, associated with the header files contained in the `SIO2_SDK/inc` directory.

- `SIO2_SDK/tools`: All the necessary tools that you need to get you started are located inside

this directory. You can also find other links to optional external tools and other goodies, located in sub-directories, that will help you to develop your games using SIO2.

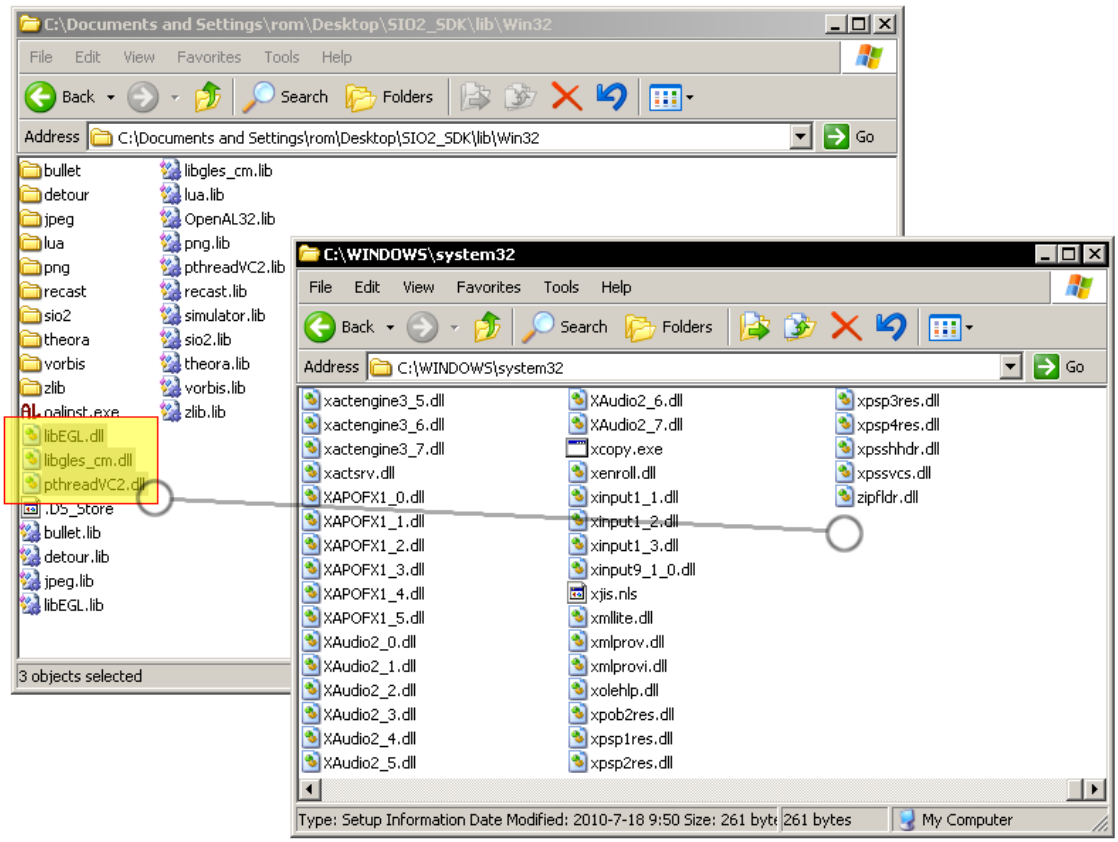
- `SIO2_SDK/tutorials`: This is the best place to start learning, inside this directory you have access to multiple tutorials that will guide you step by steps through the different functionalities and features of the engine so you can create your own state of the art 2d or 3d game! Please take note that all source files and all 2d and 3d assets, along with the original models and texture files are also available to you in order to rebuild from scratch each tutorials.

Final Requirements

Please take note that in addition to the `SIO2 SDK`, if you are a iOS developer you will also need to download and install the latest iOS SDK, you can download it along with XCode at the following address: http://developer.apple.com/, and for Windows developers all you need is Microsoft Visual C++, the Express (free) version of the software is available at http://microsoft.com/express/Windows/ . For Android developers, make sure that you have downloaded and install the Android SDK (http://developer.android.com/sdk/index.html) and NDK (http://developer.android.com/sdk/ndk/index.html), Eclipse (http://www.eclipse.org/downloads) with the ADT plugin (http://developer.android.com/sdk/eclipse-adt.html), and to be able to compile directly within Eclipse applications that uses JNI through the NDK, download and install the Sequoyah plugin (http://www.eclipse.org/sequoyah/documentation/native_debug.php). To be able to compile and debug native code.

In addition, for Windows developer, you will also need to install the necessary dynamic link libraries (DLLs) that are required by SIO2 to be able to “simulate” a GLES mobile development environment on your Microsoft Windows.

Theses DLLs can be found under the `SIO2_SDK/lib/Win32` directory, and require you to install them before you can start developing.



To install them, simply select them all, then drag and drop them to your `C:\Windows\System32` or `C:\Windows\System` folder depending on the Windows version that you are using.

The next step is to install OpenAL, to install it, simply double click on `oalinst.exe`, also located under the `SIO2_SDK/lib/Win32` folder, then simply follow the installation instructions on screen.

For all developers, you also need to download and install the latest version Blender (http://blender.org), which is by default the 3d software used by the SIO2 SDK since it is free and available on

multiple platforms.

In addition, in order to be able to export your Blender scene to the SIO2 file format you are require to have Python installed, in order to run the Blender Python SIO2 Exporter. If you do not already have Python setup on your machine you can get a free copy by visiting the ActiveState website and download ActivePython (http://www.activestate.com/activepython).

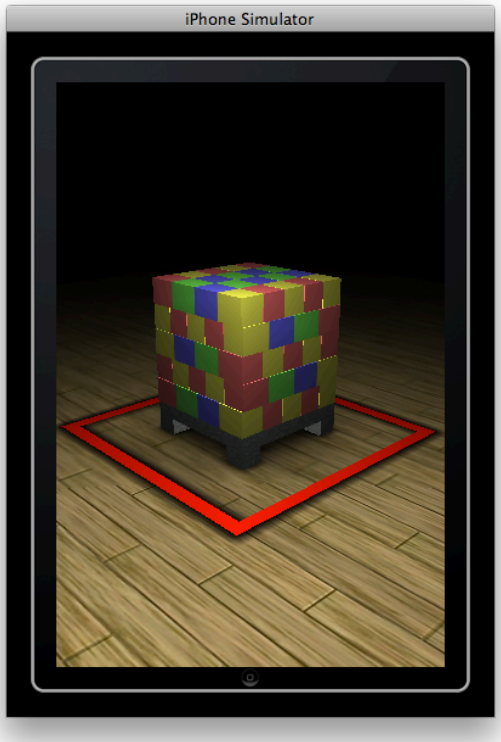
For more information about either Blender or Active Python please refer to their respective website for more detail about the installation procedure for your targeted platform.

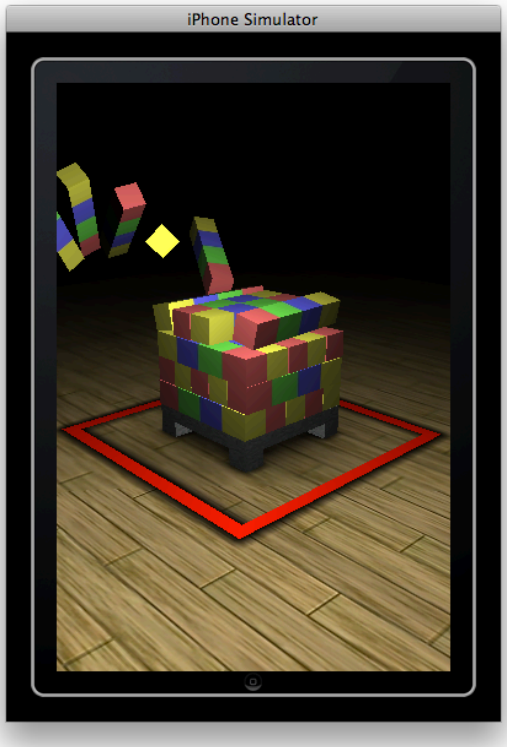
Please take note that SIO2 also support Maya and 3d Studio Max, for more information about how to use the exporter for theses software visit: http://api.sio2interactive.com.

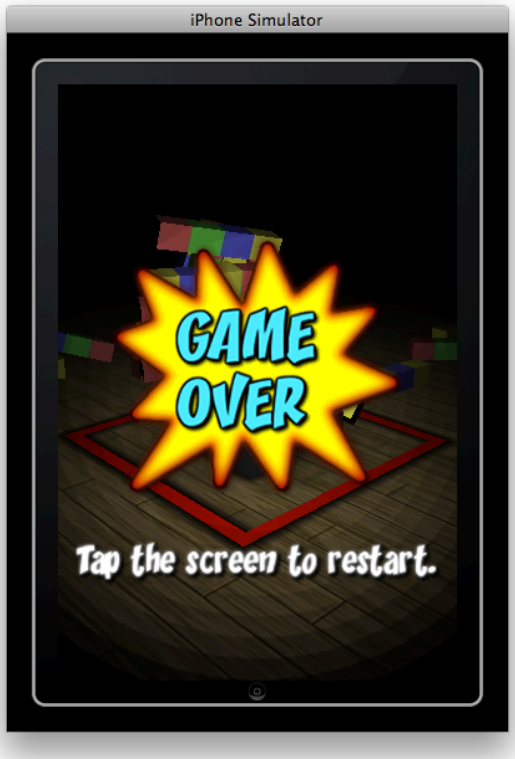
Now that your game development environment is all set! You are now good to go and ready to start creating games using SIO2!

MyGame

For this tutorial you are going to create a simple 3d casual game. Even if the game may look pretty basic at first, it touch base with a lot of different notions that you are going to use in your everyday game development.



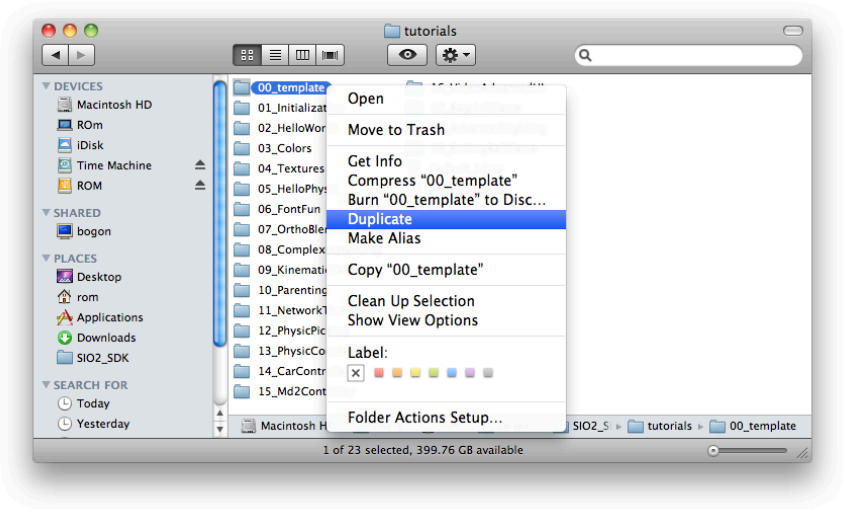




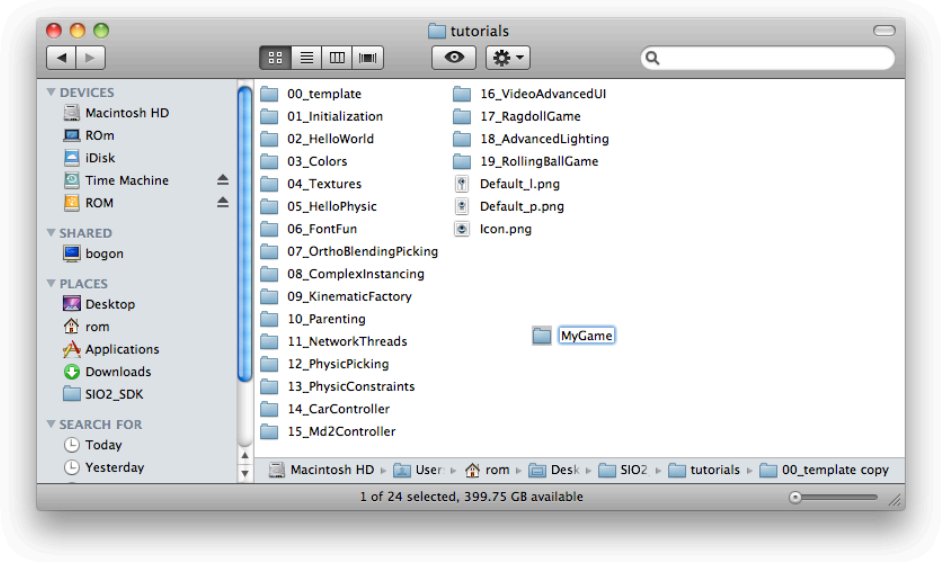
The goal of the game is to clear the table by dragging every block using the touch screen out of the red delimitation. If there's no more block on the table or if a block fall inside the red area, the game is over and the player have to touch the screen to restart the game.

Creating a New Project

For all developers, the first step before we start creating our game we need to create a new project. For this go in the `SIO2_SDK/tutorials` directory and create a copy of the `00_template` folder:



then and rename it to `MyGame`.



Then double click on it and either select the iOS or Win32 depending on which platform you are working on and open the associated project file for either XCode or VC++.

For XCode Developers

Enter the iOS directory and double click the “Untitled.xcodeproj” to launch the XCode. Now its is time to rename your project so that your App. bundle name and the executable name can be set to MyGame. Using XCode 4:

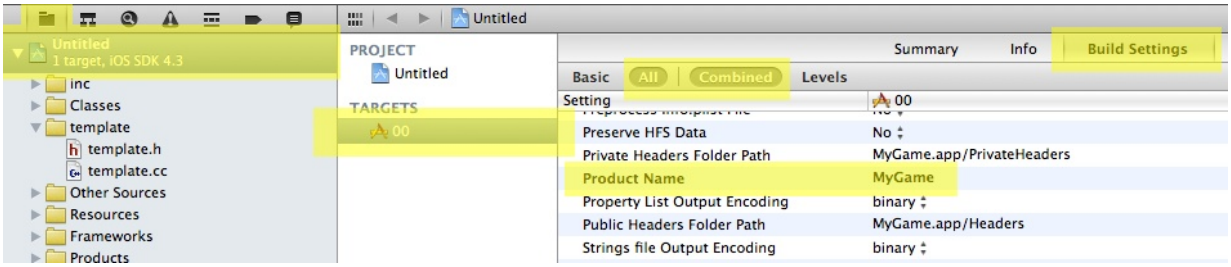
1. Select project navigator, then the current project named “Untitled”.

2. Click on the 00 inside the Targets list.

3. From the tabs on the top right select “Build Settings”.

4. Make sure that “All” and “Combined” is selected.

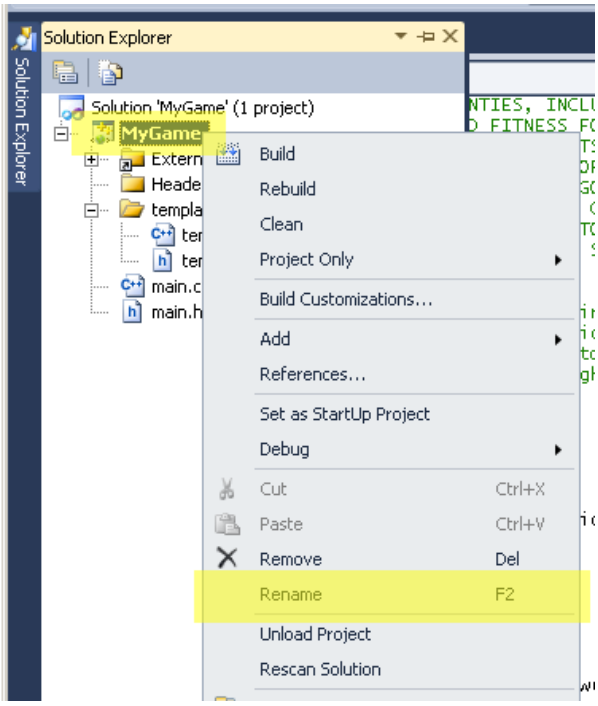
5. Browse the list to find the “Product Name” entry and change it from “00” to “MyGame”



For VC++ Developers

First double click on “MyGame.sln” to load the project into Visual C++.

Once your project is loaded open the Solution Explorer and select the `Untitled` project target, then right click on, select `Rename` from the pop up menu list, and update the field to use the name `MyGame`, just like in the figure below:

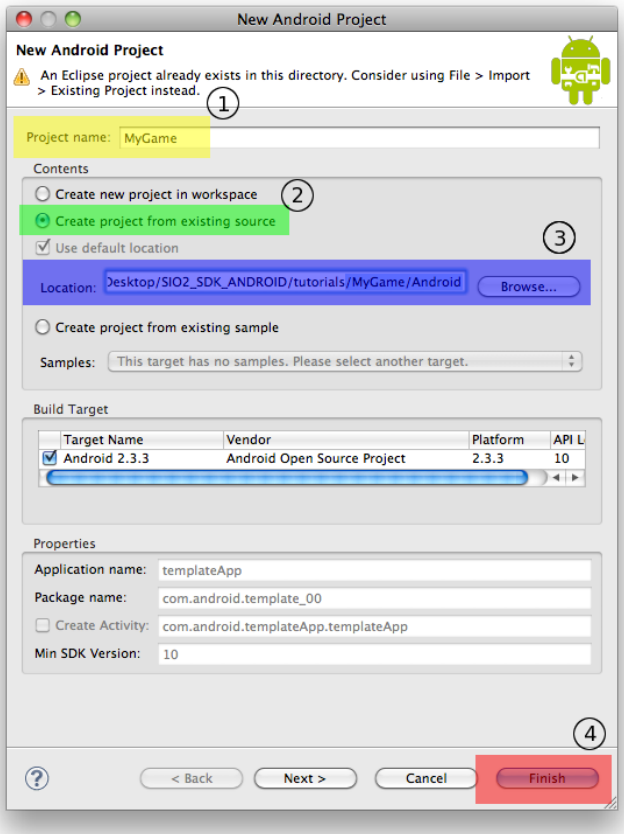


For Eclipse Developers

First you have to import the project into the Eclipse workspace. In order to do that first start by clicking the  button, to launch the wizard to create a new Android project.

Enter “MyGame” as the project name then select the radio button “Create project for existing source”. The click the browse button and select the `SIO2_SDK/tutorials/MyGame/Android`

directory. Then click the finish button.



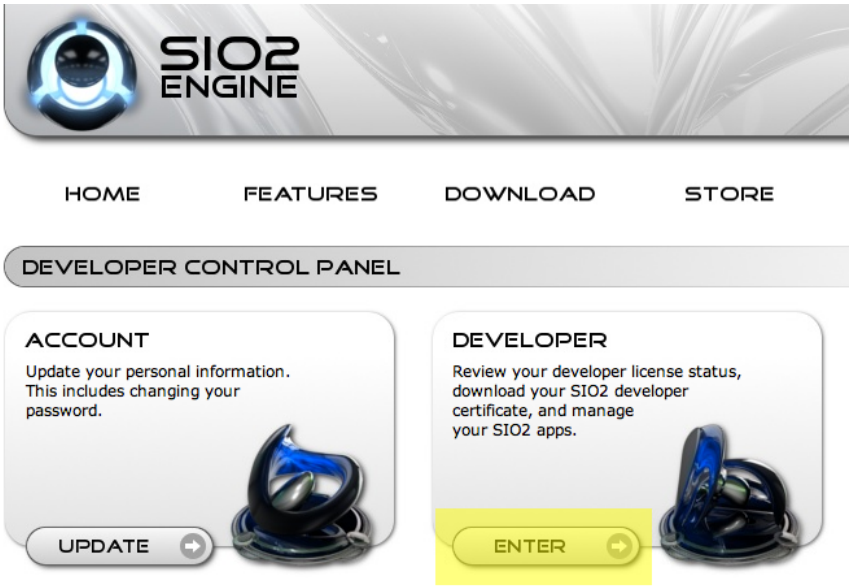
Please take note that if you want to to import other SIO2 Tutorials within your Eclipse Android development environment you'll have to repeat this procedure for each and every one of them. Or you can simply use the command like with `ndk-build` to compile and launch your application on your device. In addition, as a reminder, SIO2 and its tutorials **CANNOT** work properly on the Android simulator due to the lack of proper support for OpenGL ES v1.

For XCode, Eclipse and VC++ Developers

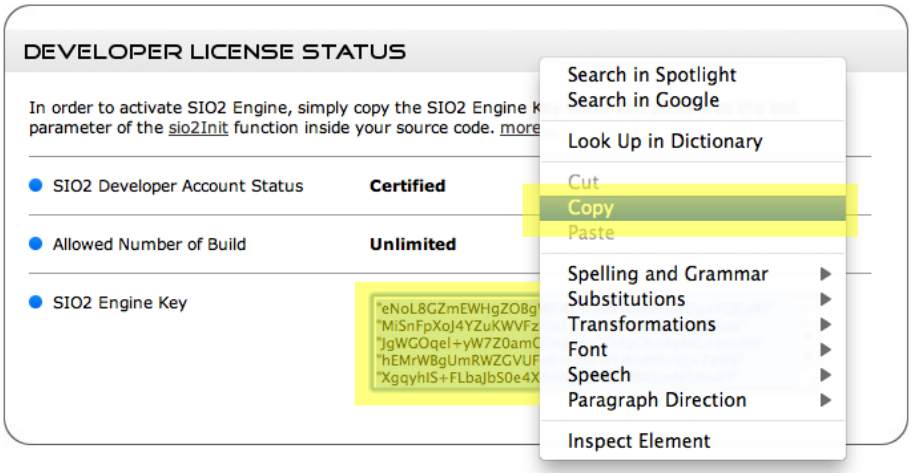
Expand the tutorial directory located inside the project tree view, and double click on `template.cpp`, this the source file that you are going to add the source code necessary to create your first game with SIO2.

As you might have already notice, the basic initialization is already created, so have a first shot at it and compile the project, but before you do so make sure that you enter your SIO2 Engine Key associated to our SIO2 Developer Account by passing it in parameter to the `sio2Init` function.

To find your SIO2 Engine key simply open your web browser and go to http://sio2interactive.com, then use the login form (the link is located at the top right of page header), then log in using your SIO2 Developer Account email and password. Once logged in, you will automatically be redirected to your Developer Control Panel, from there clock on the Developer section.



Inside this section you can find your SIO2 Engine Key located under the Developer License Status box as demonstrated below. Then simply select the whole text box content and copy it to your clipboard.



Then paste it as the last parameter of the `sio2Init` function which is located inside the `template.cpp` file:

```
51
52  #include "template.h"
53
54  void init( int _argc, char **_argv, int _width, int _height )
55  {
56      sio2Init( &_argc, _argv, "eNoL8GZmEWHgZOBgWGKUbuu0VuybBQMDgx4QGzKI"
57                               "MiSnFpXoJ4YZuKWVFzt5eLgmhZfn04YZhCaGOGan"
58                               "JgWGOqel+yW7Z0am0OeEujgGOdqGhnAyMG+eec0H"
59                               "hEMrWBgUmRWZGVUFe6/n7I63PLWivHfr/oj+7z99"
60                               "XgqyhIS+FLbaJbS0e4XfH0be1N8NYfM1mM5XsvFf"
61                               "+p33NGlzw50Ab3YOZBcFeDMyiTPjci0vAwgwAvGS"
62                               "RhCLDLezQtweWgGyi5UNYh4jQw6QXgM23SiJh/ua"
63                               "qcqLHZMsj835bz6pTSJ5Rsm/B6GMHUrlf15+0PP+"
64                               "U+HmkikoxZXEISUoJMbIzevLreR0uvn0218nrr36"
65                               "pvTc0EVqoh9L86StQlfnGc+bv51R1NLlwV7+szuu"
66                               "L1l87ng3A/fSSZErxV7Haxz8Gx0/Vzhxk+/SSdxb"
67                               "Eg73sKZFn/kUc336O0NHFtmFuvfmd9rZzJn/7qVL"
68                               "45efD3e0Ppd/uFnQtnulzp7XBjd74rZrP/+a0Twl"
69                               "PPfmjaRLu6QjFlpZhvqUP/i6eM7DK2W5RnozJVWY"
70                               "5u6M5J5klJvypmNv3+Hvj/nzDrMKJTx7di1EkuWd"
71                               "scaHiw8975RUe93IuDzt27TeW8t+z13N9w8Az3rW"
72                               "nw==" );
73
74      sio2InitGL();
75
76      sio2SetWindow( sio2WindowInit() );
77
```

In addition, make sure that as long as your SIO2 App. is in development you are connected on the Internet in order for SIO2 to be able to communicate with the SIO2 Authentication Server in order to validate your SIO2 Engine Key every time you successfully start your App.

However be re-assured that the final version of your game will not effectuate no further check, and will automatically authenticate your SIO2 Engine Key locally and print on the device system console your SIO2 Developer information. Now compile and run the project!

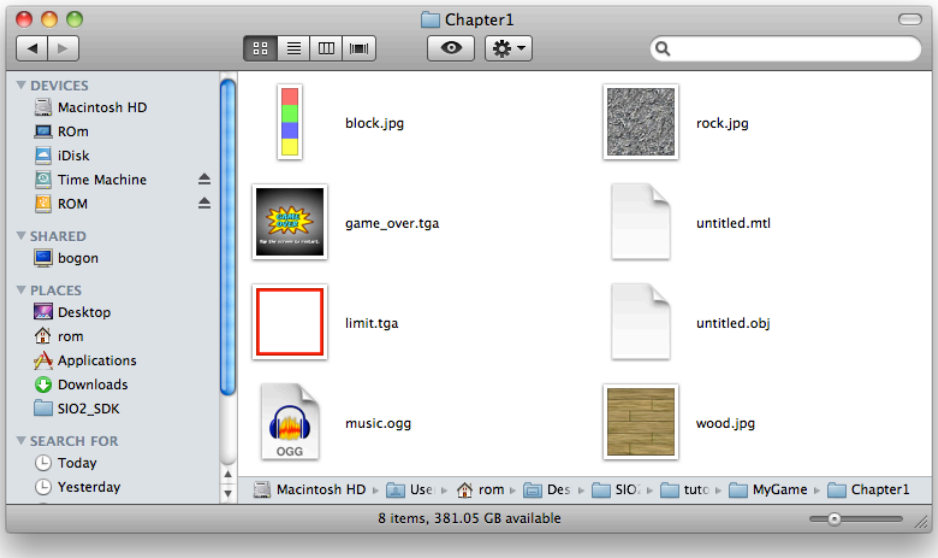
If everything went fine, you should have a the simulator window up and running, displaying a black background, which is perfectly “Ok” since this means that SIO2 is running at full speed in the window, from there we will now start adding code to start drawing on the screen.

If something goes wrong and your App. will simply exit, don’t panic its not a crash, check the system console to find more information about what might have cause SIO2 to exit.

The Scene

For this getting started tutorial you will use Blender to build your SIO2 Scene, the first step is going to be to re-create the 3d Scene for your game as demonstrated in the introduction.

All assets necessary to re-create the scene from scratch for this chapter are available inside the `SIO2_SDK/tutorials/_MyGame/Chapter1`:



Now its time to import the scene into Blender!

1. Open Blender.

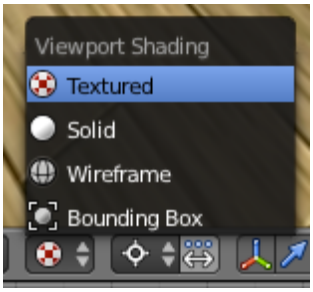
2. Select the default cube by right-clicking on it. Then push X to delete it and confirm the operation.

3. From the Blender top menu select File > Import > Wavefront OBJ, then go select the `untitled.obj` file located inside the `SIO2_SDK/tutorials/_MyGame/Chapter1` directory.

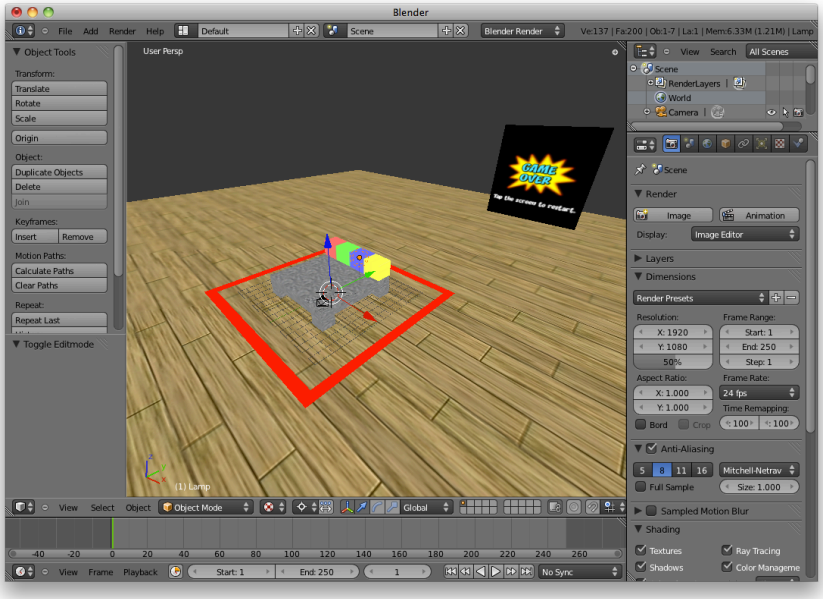
4. Then select all the objects by pressing A (once or twice) on your keyboard while your mouse is over the 3d viewport, to select all the objects.

5. Then from the Object menu select Transform > Origin to Geometry, this action will reset the pivot point of all the selected meshes to the center of their bounding box:

6. Now switch to texture view to be able to get a preview of the current scene the way is going to look like in SIO2 by selection Textured from the Viewport Shading list box:



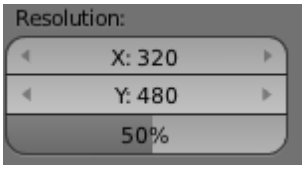
You should now see the following inside the Blender 3d viewport:



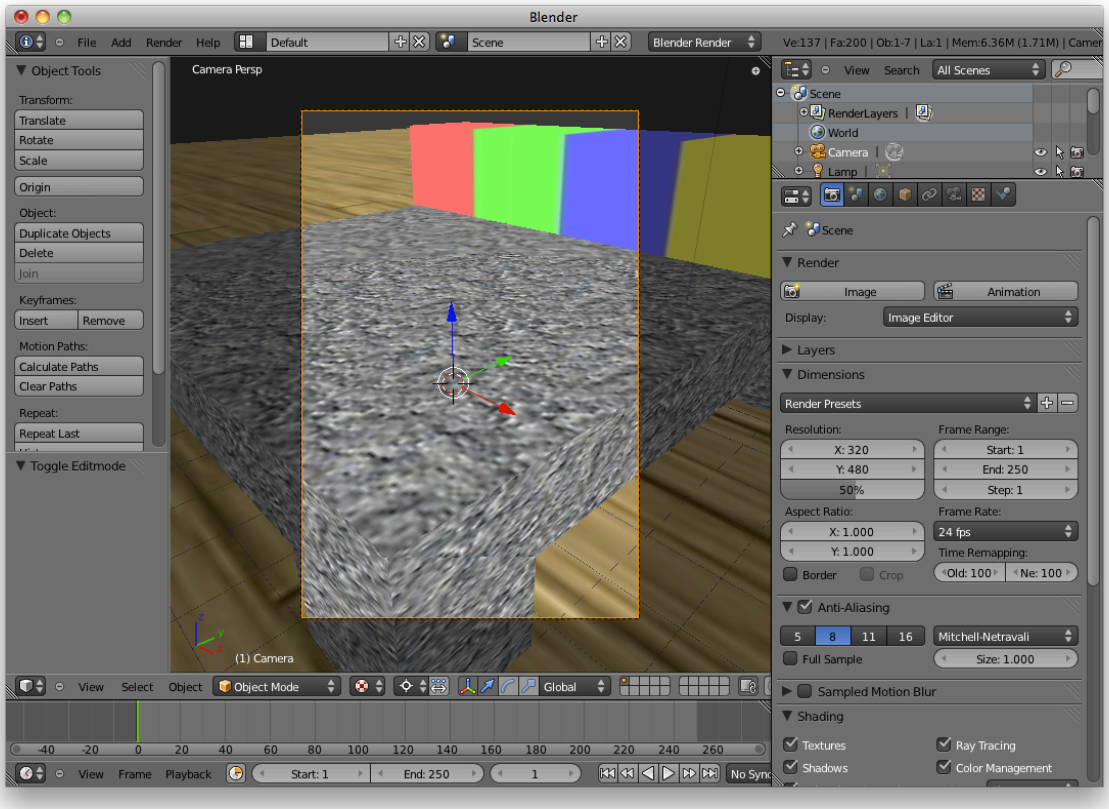
Fixing The Camera

Now its time to adjust the resolution and to place the camera into space in a way that it can now visualize all the scene.

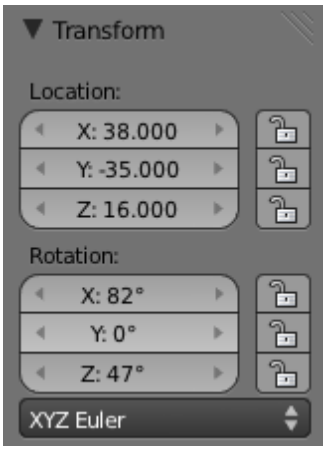
1. On the right panel select the Render icon () and change the Resolution to be 320x480 like in the figure below:



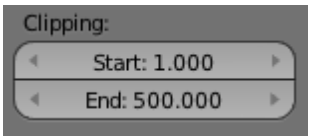
2. Next select the camera by right-clicking on it and push 0 on your keyboard to make it active in order to be able to look at the scene from the camera perspective inside the Blender 3d viewport, like in the screenshot below. Please take note that the orange line around the visible viewport camera, since you have change the resolution above they now perfectly fit the way it will be displayed on your iPhone.



3. As you can the default camera position and rotation doesn't match with the bound of the scene. To move it to an appropriate location hit Shift+F on your keyboard to enter first person view mode. Then using WASD on your keyboard move the camera into a location that the whole scene will be displayed. But not the "game over" plane. A suitable location and orientation should be obtain by moving the camera around the following values:



4. You might have probably notice that when moving the camera the far clipping plane was actually cutting the ground plane. To fix this issue, simply select the  camera data button and modify the Clipping values to be 1.0 for the near plane and 500.0 for the far plane:



Renaming The Objects

In order to have access within your game to some more “friendly” and representative names than the one used by default by the Wavefront OBJ importer. Rename the objects like in the table below so SIO2 will use theses names internally.

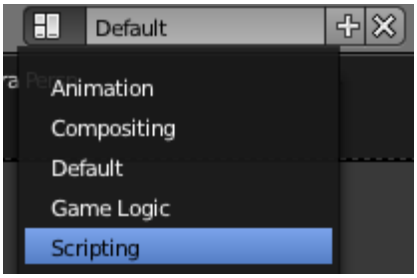
To rename an object simply toggle the object panel by clicking the  button then rename the unique data block ID name.

Old Object Name	**New Object Name**
Mesh	floor
Mesh.003	table
Mesh.004	limit
Mesh.002	game_over
Mesh.001	block

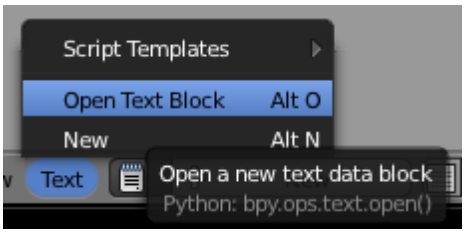
Export

Time to export the scene your scene for the first time in order to make sure that what you are seeing in Blender is exactly what you are going to get inside the iOS Simulator on your Mac or the iDevice Simulator on Windows, in order to do that follow the steps below.

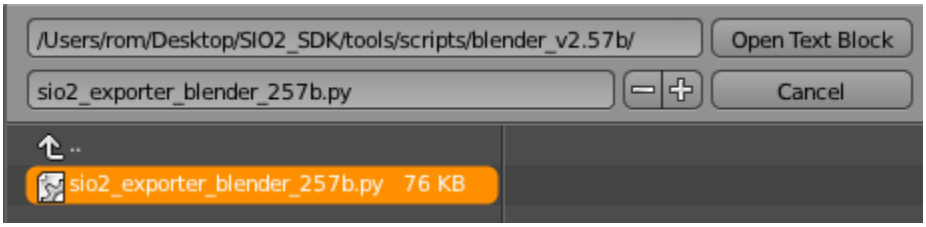
1.Since its the first time you are exporting change the Blender Screen Layout to Scripting:



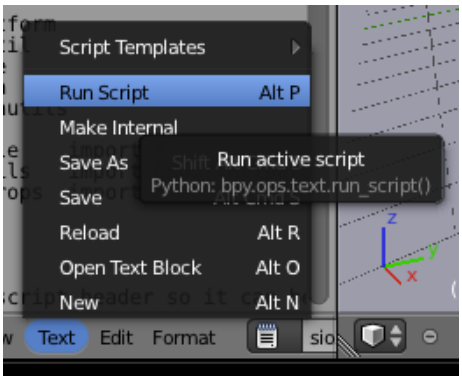
2. Then from the Text Editor window click Text > Open Text Block



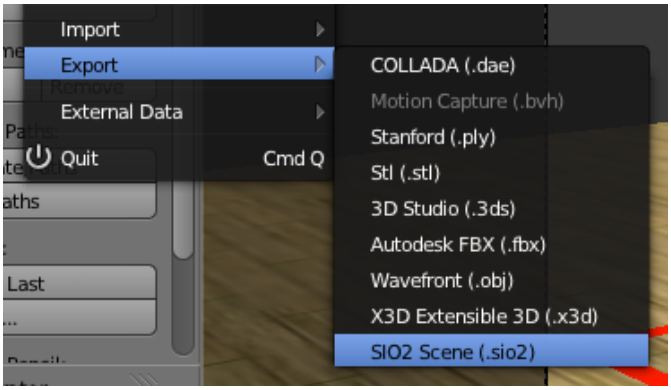
3. Next load the the SIO2 Python Exporter Script for Blender v2.57b file `sio2_exporter_blender_257b.py` located under the `SIO2_SDK/tools/scripts/blender_v2.57b/` directory, then click the “Open Text Block” button to load it inside the Text Editor.



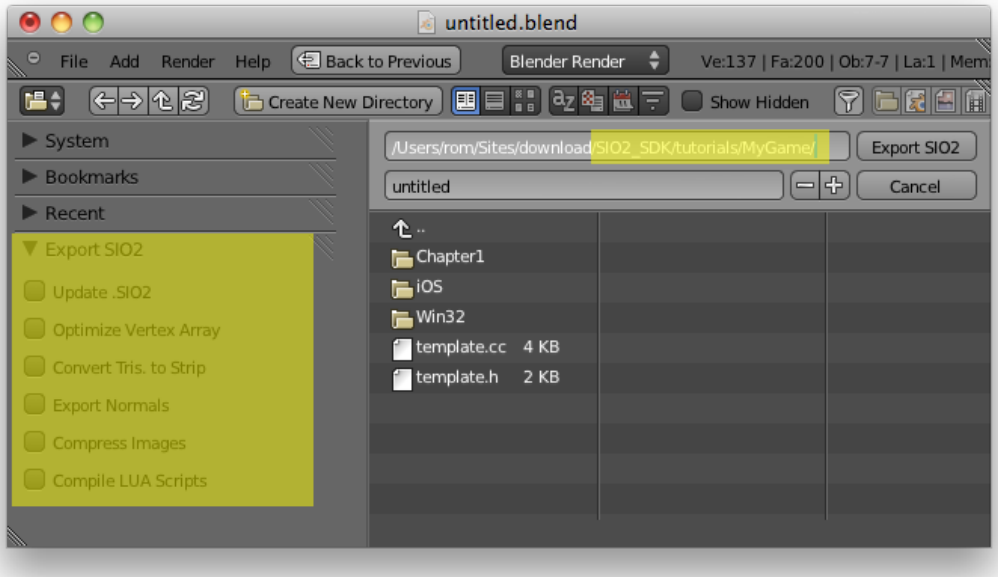
Now run the script once to make it part of the Blender File > Export Menu, to run the script simply select from the Text Editor menu Text > Run Script (or simply push ALP+P) to run the active script.



4. You can now have a direct access to export your scene using the Blender menu:

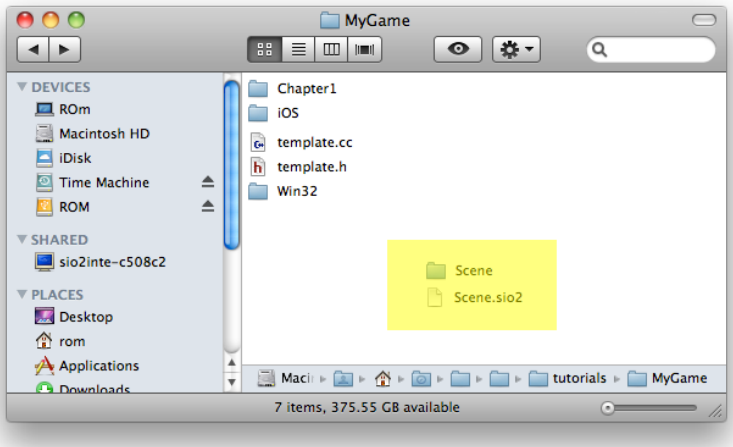


5. Now time to export, to do this simply click A (once or twice) to select all the objects in your scene. When they are all selected simply use the Blender > File > Export > SIO2 Scene menu to trigger the exporter interface:



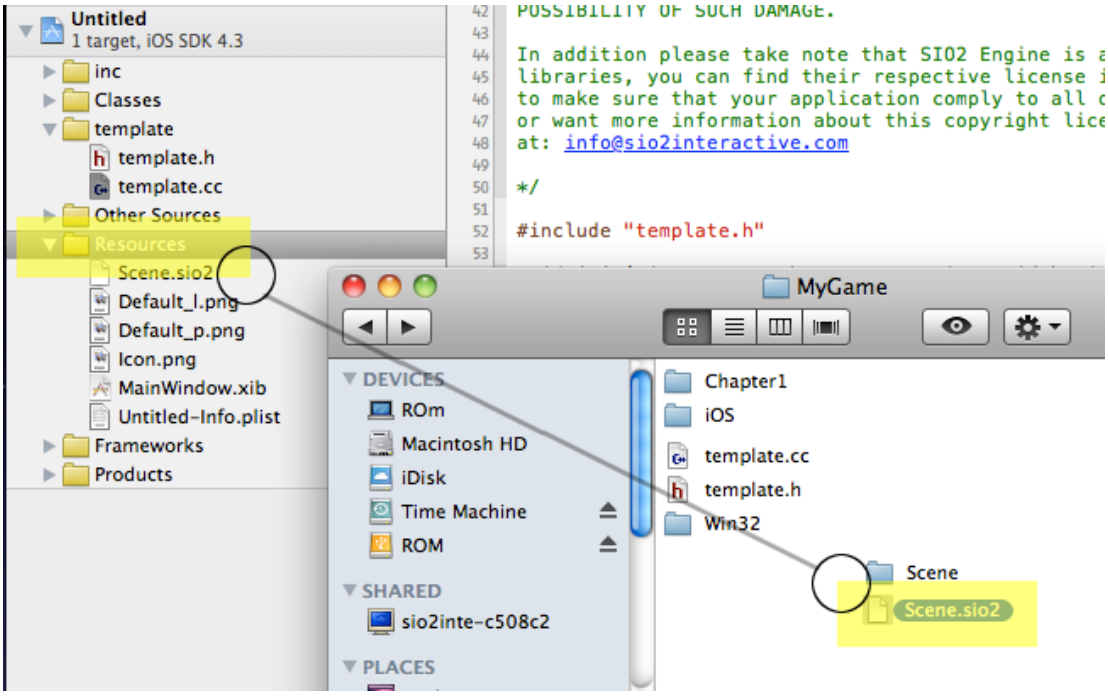
As you can see from the screenshot above the exporter options are available on the left side, however do not toggle any of them at the moment, what you are trying to do now is to simply export the current scene data to the `SIO2_SDK/tutorials/MyGame` directory to be able to create the `.sio2` for your scene and then be able to link it with your project.

6. Optionally, you can now if you open the `SIO2_SDK/tutorials/MyGame` directory with your file explorer you will notice that a directory name “Scene” have been created along with a file named “scene.sio2”. This .SIO2 is basically a ZIP file that contain all the data from the Scene directory:



For XCode Developers

It is now time to link the `Scene.sio2` file to your project in order to be able to load it at initialization time. To do this if you are using XCode simply drag the file from its location in Finder to the `Resources` folder inside the XCode project tree, as demonstrated in the following screenshot.



For VC++ Developers

On Windows no extra operation needs to be done, you are going to later on load the .sio2 file in code taking in consideration the relative location of the .SIO2 scene file directly in code.

For Eclipse Developers

Copy the Scene.sio2 file and paste it within Eclipse inside the assets directory from your project architecture. Please take note that this operation have to be done everytime you output a new version of the scene.

Compile

It is now time to have a first test run with the Scene, in order to do that you must first load the .SIO2 from disk then and render it using the SIO2 Engine API.

Loading and Rendering

Now save your Blender file inside the `SIO2_SDK/tutorials/Chapter1` directory using the File > Save dialog, in order to have a backup of this files before modifying other properties.

It is now time to start coding! Let's open template.cpp (located on your project solution

architecture), and locate the `init` function, which currently contain our basic SIO2 initialization. First we need to define where your `.sio2` is located for that we are going to add the following code above the `#include` call.

```
#ifdef SIO2_WIN32

	// On Windows, go one directory down to find the .sio2
	#define TUTORIAL_SCENE "../../Scene.sio2"
#else
	// On Mac, our Scene.sio2 is embedded inside the .app and
	// is located at the same level of our executable.
	#define TUTORIAL_SCENE "Scene.sio2"
#endif
```

Then insert the following code right after the SIO2 OpenGLES initialization function `sio2InitGL`.

```
// Initialize a new SIO2resource pointer call it “default” and make it available
// globally, using the sio2SetResource function.
sio2SetResource( sio2ResourceInit( "default" ) );
{
	// Declare a counter.
	unsigned int i = 0;

	// Create a dictionary for the SIO2resource to be able to load
	// and parse our 2D and 3D assets.
	sio2ResourceCreateDictionary( sio2GetResource() );

	// Open the .sio2 file in read only mode.
	sio2ResourceOpen( sio2GetResource(), TUTORIAL_SCENE, 1 );

	// Loop while we have some entry inside our .sio2
	while( i != sio2ResourceGetNumEntry( sio2GetResource() ) )
	{
		// Extract, parse and dispatch the stream to the appropriate loader.
		sio2ResourceExtract( sio2GetResource(), NULL );

		  // Next entry.
		 ++i;
	}
	// Close the .sio2 resource.
	sio2ResourceClose( sio2GetResource() );

	// Link all the materials to their respective objects.
	sio2ResourceBindAllMaterials( sio2GetResource() );

	// Link all the images to their respective materials.
	sio2ResourceBindAllImages( sio2GetResource() );

	// Pre-calculate the localspace and worldspace matrix for every objects.
	sio2ResourceBindAllMatrix( sio2GetResource() );

	// Generate all the OpenGL VBO and Texture ID.
```

```
        sio2ResourceGenId( sio2GetResource() );

        // Reset the SIO2 states.
        sio2Reset();

        // Reset the GLES states maintained by SIO2.
        sio2StateReset( sio2GetState() );

        // Get the first camera in our SIO2resource and make it accessible
        // globally using the sio2SetCamera function.
        sio2SetCamera( ( SIO2camera * )sio2GetResource()->_SIO2camera[ 0 ] );
}
```

Now all you have to do is to include the following rendering code inside the `draw_frame` function:

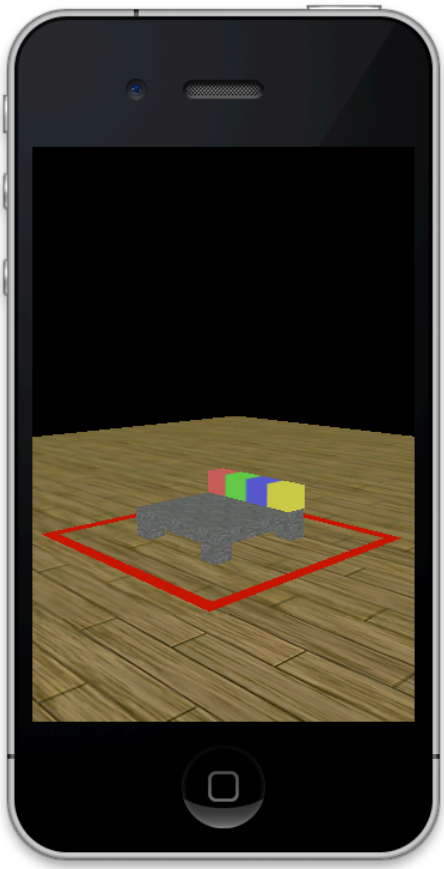
```
void draw_frame( void )
{
     // Clear the depth buffer and the color buffer.
     glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );

     // Select the projection matrix.
     glMatrixMode( GL_PROJECTION );

     #ifndef SIO2_WIN32

          // If we are on Mac, reset the projection matrix.
          glLoadIdentity();

     #else

          // For Windows user, its impossible to "flip" the iDevice Simulator
          // since the OS doesn't allow us to do that, so the iDevice Simulator
          // maintain its own projection matrix based on the orientation set
          // at initialization time, so we don't need to reset the projection
          // matrix.
     #endif

     // Set the perspective of our 3D viewport using the information of
     // our current camera previous set inside our init function.
     sio2Perspective( sio2GetCamera()->fov, // The field of view in degree.
                      sio2WindowGetAspectRatio( sio2GetWindow() ), // Get the aspect
ratio of the viewport.
                      sio2GetCamera()->cstart,  // The start clipping plane distance.
                      sio2GetCamera()->cend, // The end clipping plane distance.
                     ( float )sio2GetWindow()->orientation );  // The viewport
orientation in degree.

     // Select the modelview matrix.
     glMatrixMode( GL_MODELVIEW );

     // Reset it.
     glLoadIdentity();
```

```
	// Multiply the current modelview matrix with our camera matrix.
	sio2CameraRender( sio2GetCamera() );

	// Get the GLES modelview matrix.
	sio2CameraGetModelviewMatrix( sio2GetCamera() );

	// Get the GLES projection matrix.
	sio2CameraGetProjectionMatrix( sio2GetCamera() );

	// Build the camera frustum based on the matrix that we just gather.
	sio2CameraUpdateFrustum( sio2GetCamera() );

	// Use the camera frustum to determine which objects are visible
	// and which one are not.
	sio2ResourceCull( sio2GetResource(),
	                  sio2GetCamera(),
	                  SIO2_CULL_OBJECTS );

	// Render all visible solid objects in our scene, and since we
	// are going to use physic we also want to keep track of the
	// current position of the off screen objects.
	sio2ResourceRender( sio2GetResource(),
	                    NULL,
	                    NULL,
	                    SIO2_RENDER_SOLID_OBJECT   |
	                    SIO2_RENDER_CLIPPED_OBJECT );
}
```

And finally, insert the code to be executed when the application exit by adding the code below to the `close_app` function:

```
void close_app( void )
{
	// Free out global SIO2resource.
	sio2ResourceUnloadAll( sio2GetResource() );

	// Free the SIO2window.
	sio2WindowFree( sio2GetWindow() );

	// Shutdown SIO2.
	sio2Shutdown();

	printf( "Shutdown...\n" );
}
```

Now build and run the program, you should now see the following on either your Windows iDevice Simulator or on your Mac iPhone Simulator:



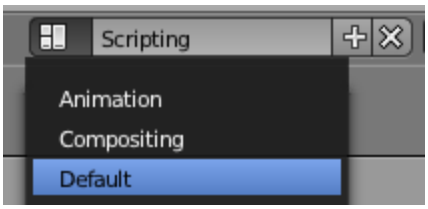
This conclude Chapter1, you can find the all the files that were used or modified in this chapter inside the `SIO2_SDK/tutorials/_MyGame/Chapter1` directory for your reference. Compare them with yours to make sure that everything is identical before you moving on to the next chapter.

2. Physic, Lighting and Instancing

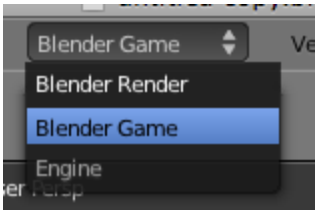
Inside the following sections you will learn how to setup the necessary properties to the world as well as your objects to support physic. In addition, you will add light to the scene and learn how to control the Lamp properties to get realistic results. And finally you will discover how to instance objects and save precious rendering time.

Physic and Instancing

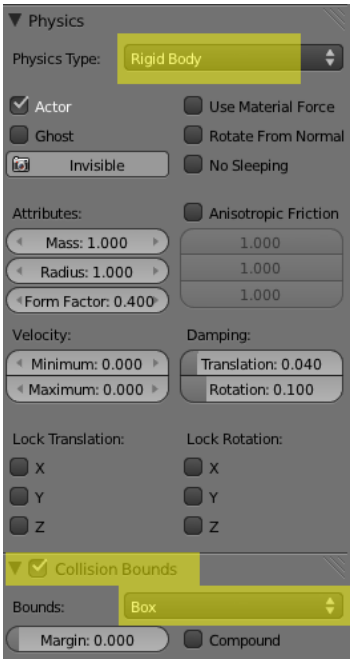
Before adding physic properties to you objects first get back to the Default screen layout, by selecting it from the Screen Layout list box located on the top bar:



Then switch the Blender Game from the Rendering Settings list box located on the top bar. It is necessary to switch it to game in order to gain access to the physic settings used by SIO2.



Now select the object named “block” by right clicking on it, the click the Physics button  to access the physic properties of the object. Inside the Physics area select from the Physic Type list box Rigid Body. Then click the Collisions Bound check box (located at the bottom of the panel), and make sure that Box is selected as the Bound Type. You should now have the same settings as in the following screenshot.



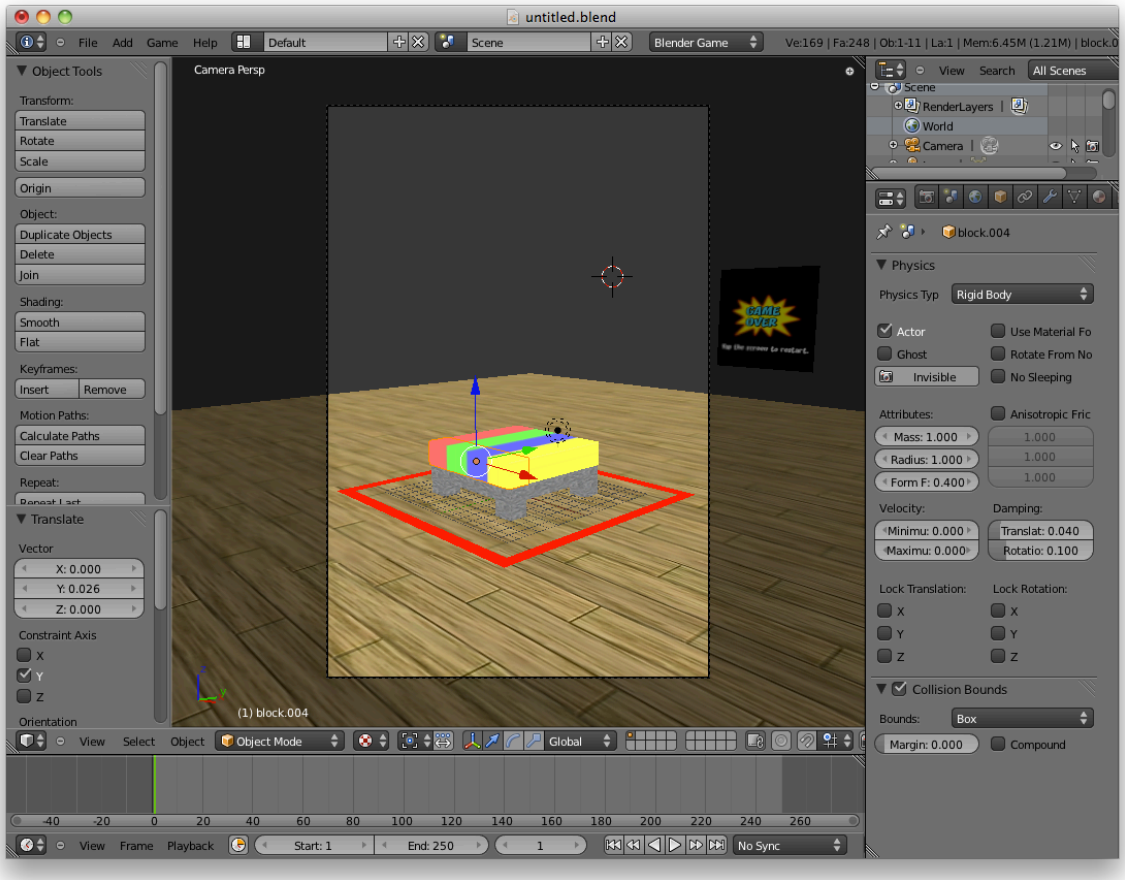
Now that you have assigned physic data it is time to instance the object. Object instancing in SIO2 drastically improve the rendering speed, what instancing does it share all the mesh data from the master to the children objects, and avoid unnecessary machine state switch by rendering the master object first then flush all the childrens without changing any states.

Please take note that instancing is name based and the exporter will search for the ".???" tag

inside the object name. For more information about instancing, please consult the Blender SIO2 Exporter manual.

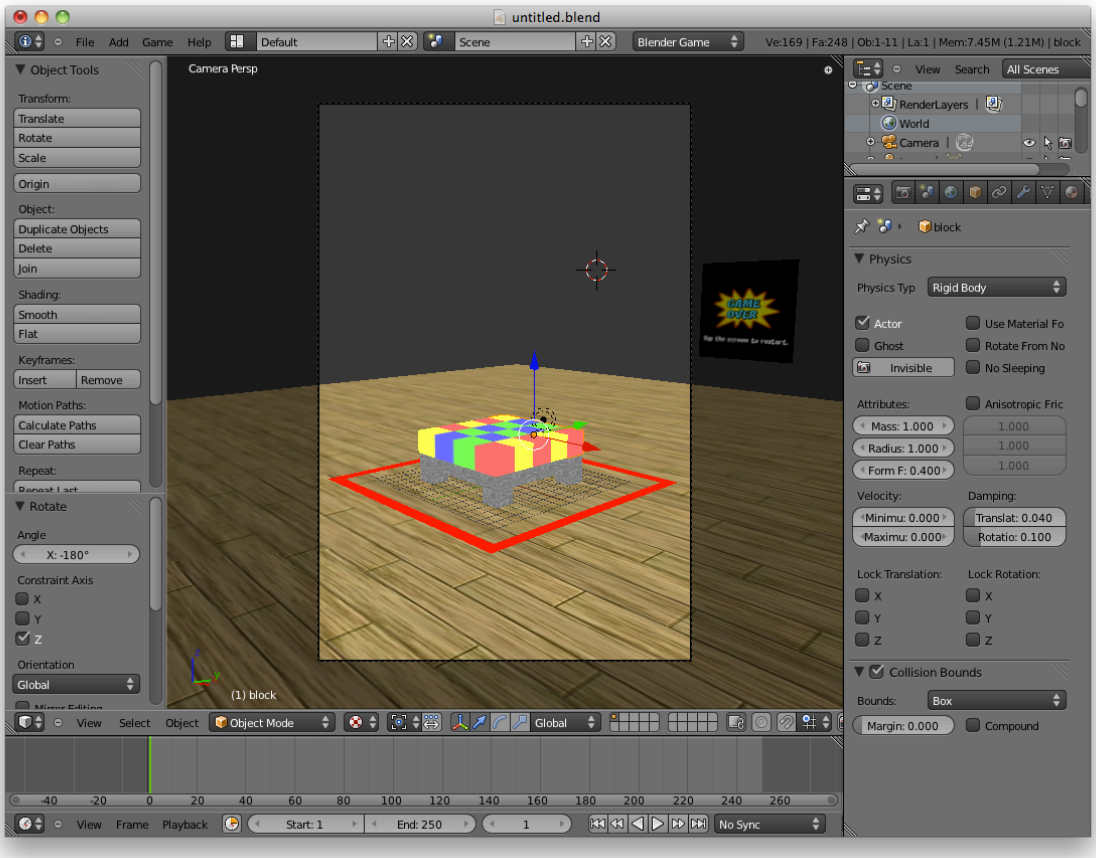
For instancing your block object, (make sure it is selected and that we are in Object Mode) then simply hit ALT+D in Blender to create a new soft copy of the object, in addition please take note that Blender automatically create a new valid SIO2 instance for each duplicated block by adding the ".???" tag to the object name.

Now use the 3D Transform Manipulator to move the block away from the first one on the Y axis (the green arrow) filling a first row of blocks on our table object like this:

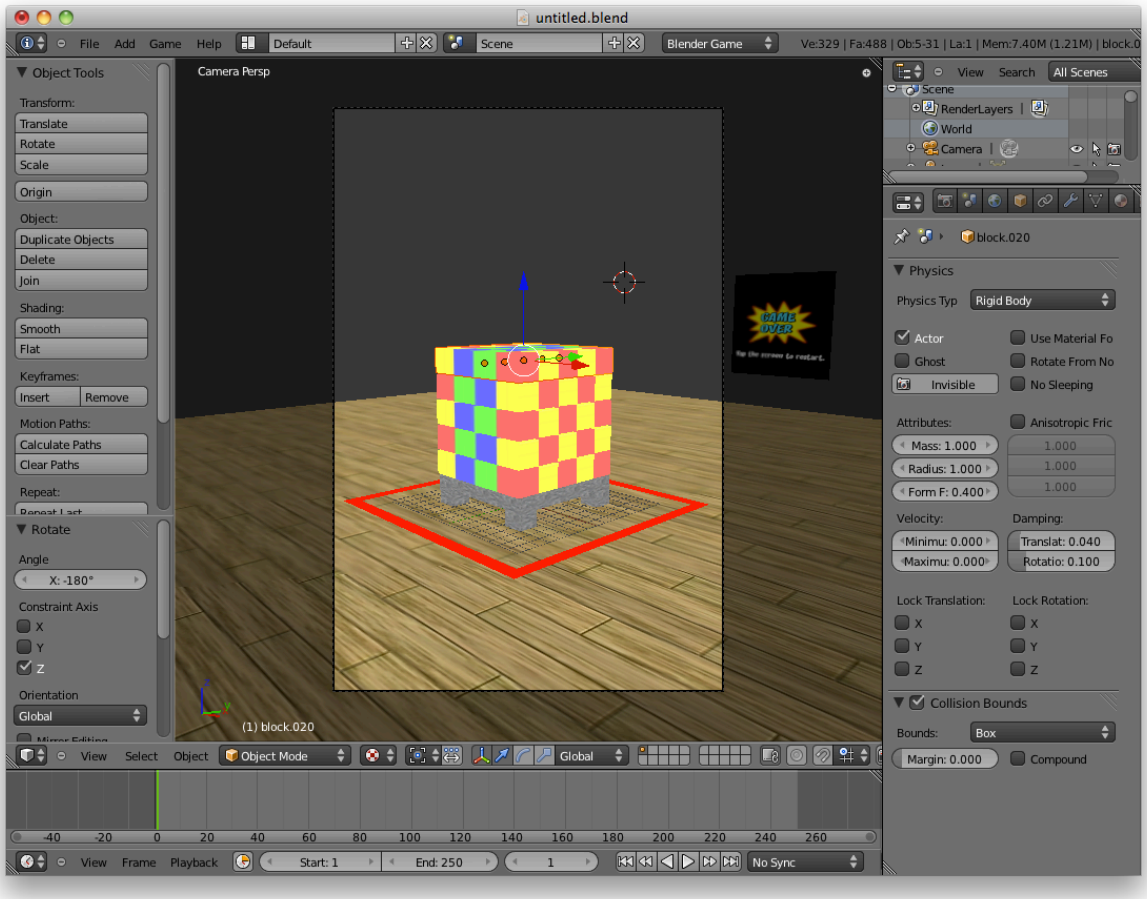


When you are moving the block make sure that they don't overlap each other and leave a small distance between each of them. If the objects are too close from each other, the physic simulation will not work properly.

If you want to get fancy an invert a few blocks just to break the linear pattern simply select a block by right clicking on it then press R, then Z and type 180, to rotate the object on the Z axis of 180 degree and create the following pattern:

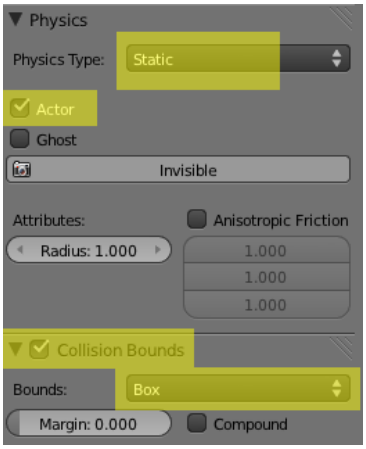


Now lets start stacking a few columns, to do that, hold the shift button on your keyboard and right click on every block one by one to make sure they are all selected. Then push ALT+D to duplicate the whole row. Then once again using the 3D Transform Manipulator offset the newly created instances up on the Z axis, and make sure they are not overlapping the objects of the first row. On the keyboard type R then Z then 180 to rotate the whole row in order to invert the texture pattern. Repeat the steps above to stack about 4 or 5 row high, you should now have something similar as the following:



Since now all your physic blocks are set we need to make sure that they won't just fall down in space since there's nothing that can support them at the moment, and if you start the physic simulation the block will just pass right through the table and the floor.

You now have to assign the necessary physic property to the floor and the table. To do this first select the floor by right clicking on it then get back to Physics panel by clicking the  button, if its is not already selected. Then from the Physics attributes select that the Physic type is set to Static, toggle the Actor check box, enable to the Collision Bounds check box and make sure that Box is selected, like in the figure below.



Now repeat the same operation as you did for the floor but this time for the table and the limit object. Once it is done, select all the objects in the scene, the select File > Export > SIO2 Scene (.sio2) and set that the scene output directory is the same as where you export your scene before basically located inside the `SIO2_SDK/tutorials/MyGame` folder as you did previously. Now before clicking the Export button make sure that the 

is checked so you won't have to re-export the scene from scratch but simply update it, then export.

Create and Render a Physic World

Now get back inside the `init` function callback inside `template.cpp`, then locate the following line:

```
sio2ResourceBindAllMatrix( sio2GetResource() );
```

and add the following code right before calling the function:

```
// Link all object instances to their master object. It is necessary
// to do that BEFORE we pre-calculate their matrices.
sio2ResourceBindAllInstances( sio2GetResource() );
```

As you might have guess, since now you have some objec instances you have to make sure that they are all bind to their respective master before pre-calculating their matrices. Now insert the following code after the sio2ResourceBindAllMatrix function call:

```
// Create a new physic world and set it to be globally available
// using the sio2SetPhysic function.
sio2SetPhysic( sio2PhysicInit( "world" ) );

// Set the physic state to SIO2_PLAY.
sio2PhysicPlay( sio2GetPhysic() );

// Link all the physic object to the globally set physic world.
sio2ResourceBindAllPhysicObjects( sio2GetResource(), sio2GetPhysic() );

// Get the first SIO2world in the stack, and make it accessible globally,
// using the sio2SetWorld function.
```

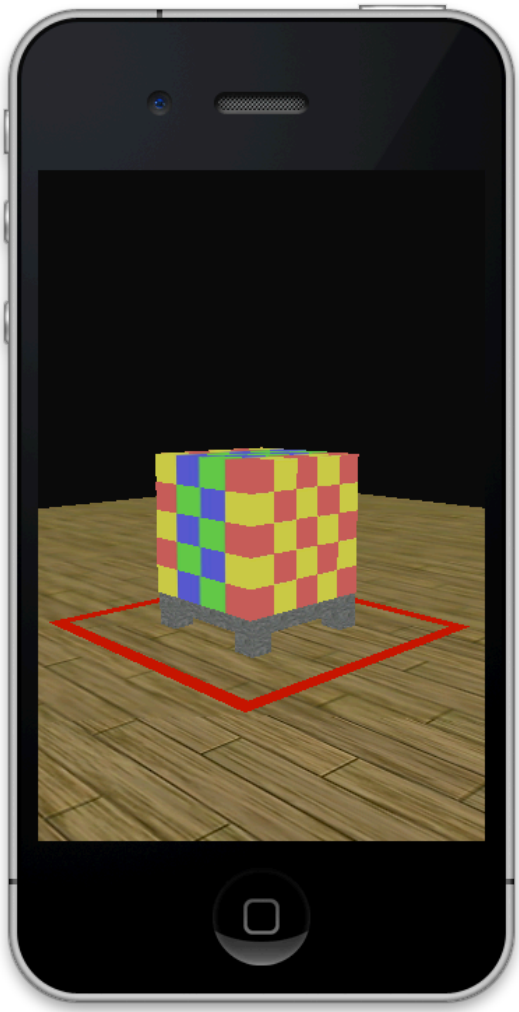
```
sio2SetWorld( ( SIO2world * )sio2GetResource()->_SIO2world[ 0 ] );

// Setup our SIO2world and pass the current sio2GetPhysic pointer which
// basically point to our newly created physic world in order to affect
// the gravity value contain in the SIO2world to our SIO2physic world.
sio2WorldSetup( sio2GetWorld(), sio2GetPhysic(), 0.0f );

// Now that all our physic object have been added to the SIO2physic world,
// run a quick optimization to sort the different physic entities.
sio2PhysicOptimize( sio2GetPhysic() );
```

Now, move on to the `draw_frame` function so you can insert inside the rendering loop the necessary code to step in the physic simulation every frame based on the delta time of your game, in order to have a continuous physic simulation synchronized with the rendering.

Then locate the `sio2ResourceRender` function call and add the following block of code on the next line right after the call:

```
// Physic simulation.
{
	// Declare a static variable to count the number of frame.
	static unsigned int fra = 0;

	// Increase the current frame number.
	++fra;

	// If we are above 10 frame (10 is arbitrary here), we should now get
	// a stable delta time, since as soon our game start the first few frame
	// are not very accurate time as it takes about a second to regulate it.
	if( fra > 10 )
	{
		// Increase the physic simulation step, based on our application delta
		// time accessible from the d_time property of our SIO2window.
		sio2PhysicRenderFull( sio2GetPhysic(), // Get our global SIO2physic pointer.
		 sio2GetWindow()->d_time,            // Get the current delta time
		 2 );                                       // Execute 2 full physic steps.
	}
}
```

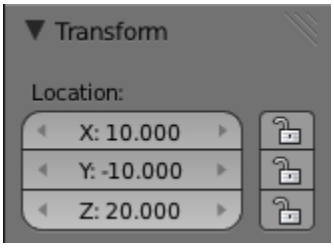
Now build and run the application. Congratulations you have successfully link instances, create a physic world and you have integrate the necessary code to run the physic simulation in real time. If you have followed the steps above properly you should now obtain the following:



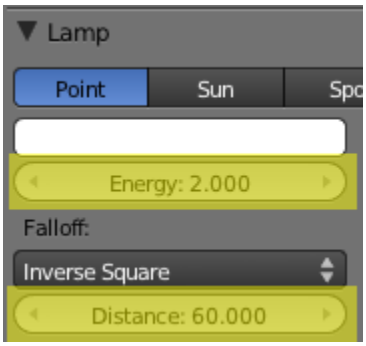
Lighting

Let's now add some light to your scene, but first go back to Blender, and right click on the Lamp (which should be hidden by the table), to pick it easily simply switch the rendering mode from Textured to Wireframe, then get back to Textured.

Move the Lamp location using the 3D Transform Tool and place it at the corner of the limit object facing the Camera, and give it some offset on the Z axis, which should be around the following coordinates:



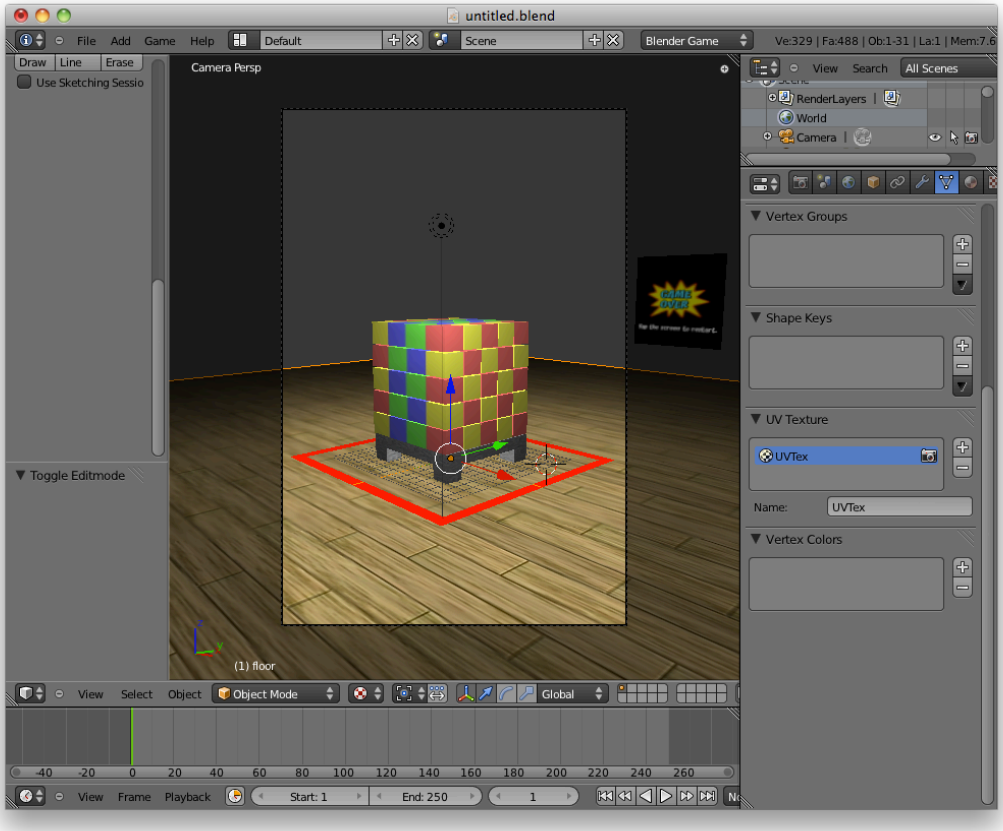
Now click Lamp data button  to access the Lamp properties and modify the distance value to 60.0, and the energy to 2.0.



For more information about which Lamp properties are supported by the Blender SIO2 Exporter please consult the user manual, where you can get more information about each and every Lamp property.

You are now almost ready to export your Scene again in order to test the ligthing settings, but before that Im sure you would like to get a preview of how our Scene would look like inside the Blender 3D viewport. This step is actually optional and not necessary but can really you to visualize how your Scene(s) will look like when Lamp(s) are present.

For this make sure that the “Copy Attributes Menu” add-on is enabled, then select (i.e) the floor object then enter Edit Mode by pushing the TAB button, select select a face. Then click the object data button  to get access to the properties, then move on to the Texture Face panel and click the Light check box. Then invert the face selection by pressing CTRL+I then click the Copy Mode button. Repeat the operation for every objects that should receive light, basically all of them except the game over plane. You should now obtain the following result inside the Blender 3d viewport:



In addition before moving forward, please take note that the preview in Blender will always be slightly darker than what you will get in SIO2, this is because Blender is using the lighting model controlled by the GL_SEPARATE_SPECULAR_COLOR, extension which is unfortunately not available in GLES. Keep that in mind when you are testing your Scene inside the Blender 3D Viewport!

You are now ready to export your scene again, apply the same procedure as before in order to overwrite the previous Scene.sio2 so make sure that the “Update .sio2” option is toggled. In addition this time since you will need to export the normals in order for the ligthing to be able to work properly, so make sure you also toggle the 

option in order to do so.

Lighting Code

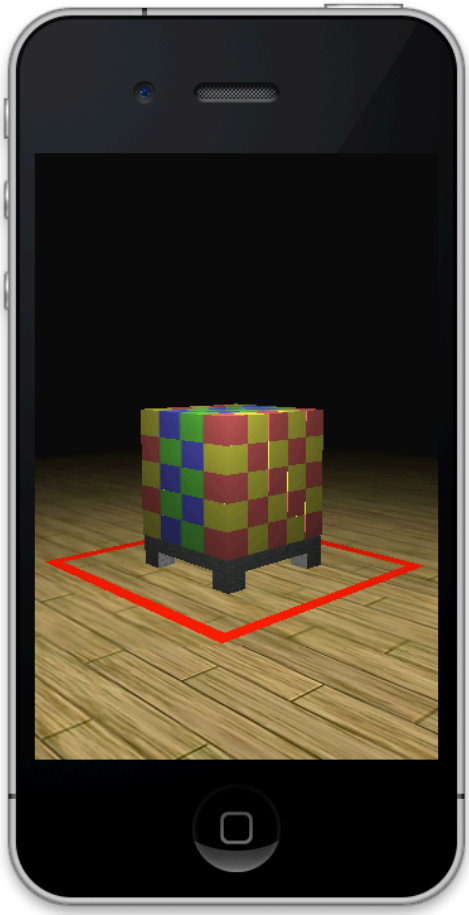
Now back to our C/C++ IDE to add more code to render the Scene with lighting enabled.

First locate the `sio2ResourceRender` call (inside `draw_frame` unction) and replace it with the following:

```
// Enable lighting and set the ambient color.
sio2LampEnableLighting();
{
	// Render all our solid objects and all the lamps available in our Scene.
	sio2ResourceRender( sio2GetResource(),
			    NULL,
```

```
                                NULL,
                                SIO2_RENDER_SOLID_OBJECT    |
                                SIO2_RENDER_CLIPPED_OBJECT |
                                SIO2_RENDER_LAMP );
}
// Disable lighting.
sio2LampDisableLighting();
```

Then build and run the application, you should now be able to see your Scene fully dynamically lighted and shaded:



This conclude this chapter, you have learned plenty enough to start doing some experimentation before moving on the next and final chapter of this getting started tutorial. In order to by sure that you have all the good properties set and the proper source code integrated, do not hesitate to compare your files with the ones located inside the `SIO2_SDK/tutorials/_MyGame/Chapter2` directory.

Chapter 3: Events and Game Logic

At this stage you are basically done with the Blender operations, your Scene is not fully set and it is now time to integrate the bits and pieces of code to finish our simple game. In this chapter you will learn how to handle events, do 3d picking and handle the game logic.

Handling Events and 3D Picking

First you have to let the player be able to drag each block on screen, for this you are going to need to create a full screen `SIO2sprite` to handle the global `touchesBegan`, `touchesMoved` and `touchesEnd` event callbacks. But first what exactly is an `SIO2sprite`? Basically it is a 2D quad on screen that represent a region that can respond to event, in addition you can also link a material to it and render texture as well. This region can also respond to transformation such as location, rotation and scale if you desire. Now locate the `init` function declaration, and insert the following variable declaration right above it:

```
SIO2sprite *_SIO2sprite = NULL;
```

Next, spot the `sio2GetWindow()->_SIO2windowrender` (located inside the `init` function), and integrate the code below on the next line to create the `SIO2sprite` in order to link and respond to touche events:

```
// Create a new SIO2sprite pointer.
_SIO2sprite = sio2SpriteInit( "fullscreen" );
{
	// Set the X and Y scale of our SIO2sprite.
	_SIO2sprite->_SIO2transform.localspace.scl.x = sio2WindowGetWidth(
sio2GetWindow() );
	_SIO2sprite->_SIO2transform.localspace.scl.y = sio2WindowGetHeight(
sio2GetWindow() );

	// Set the width and height of the SIO2sprite area
	// that will respond to event on screen.
	_SIO2sprite->area.z = sio2WindowGetWidth ( sio2GetWindow() );
	_SIO2sprite->area.w = sio2WindowGetHeight( sio2GetWindow() );

	// Pre-calculate the localspace and worldspace transformation matrix,
	// it is necessary to calculate them before we call sio2SpriteUpdateBoundary.
	sio2TransformUpdateMat( &_SIO2sprite->_SIO2transform,
SIO2_TRANSFORM_BOTH_MATRIX );

	// Pre-calculate the rectangle coordinate on screen that will respond to the
user event.
	sio2SpriteUpdateBoundary( _SIO2sprite );

	// Assign the touchesBegan callback. Everytime the user will tap in
	// our "sensitive" area of our SIO2sprite pre defined above the execution
	// pointer will be calling this function.
	_SIO2sprite->_SIO2windowtouchesBegan = touchesBegan;

	// Same as above but for the touchesEnd event.
	_SIO2sprite->_SIO2windowtouchesEnded = touchesEnded;

	// And finally the touchesMoved event.
	_SIO2sprite->_SIO2windowtouchesMoved = touchesMoved;
```

```
	// Set the flag SIO2_SPRITE_ENABLED to our SIO2sprite to enable event
processing.
	sio2EnableState( &_SIO2sprite->flags, SIO2_SPRITE_ENABLED );
}
```

Then to make sure that your events handling code is working properly let's temporary replace the event function callbacks with the code above and lets have a test run while monitoring the system console prompt to make sure that all events are handled properly.

```
void touchesBegan( void    *_ptr, SIO2touche *_SIO2touche )
{
	// Report on the console when the touche began.
	printf("touchesBegan\n");
}

void touchesMoved( void     *_ptr,  SIO2touche *_SIO2touche )
{
	// Report on the console when the touche move.
	printf("touchesMoved\n");
}

void touchesEnded( void *_ptr,  SIO2touche *_SIO2touche )
{
	// Report on the console when the touche end.
	printf("touchesEnded\n");
}
```

Now build and run your App., and when the application is loaded simply click, move and release your left mouse button over the screen while monitoring the console prompt to receive live feedbacks from the code you've just added.

After you get the confirmation that your event handling code work properly it is now time to implement 3d physic picking code in order for the player to be able to drag and drop the blocks around. To do this replace the `touchesBegan`, `touchesEnded` and `touchesMoved` function block by simply selecting them and replace their content with the following code to handle the picking using theses 3 events callback:

```
// Temp. SIO2object pointer to the currently selected object.
SIO2object *selection = NULL;

// The start position in 3D of where we start dragging.
btVector3 start_pos;

// Standard point to point Bullet physic constraint.
btPoint2PointConstraint    *_btPoint2PointConstraint = NULL;

// Temp. variable to hold the picking distance.
float pick_dst = 0.0f;

// Helper function that determine if we already have a constraint pointer
// created, and if yes, simply delete it from memory.
void remove_constraint( void )
{
```

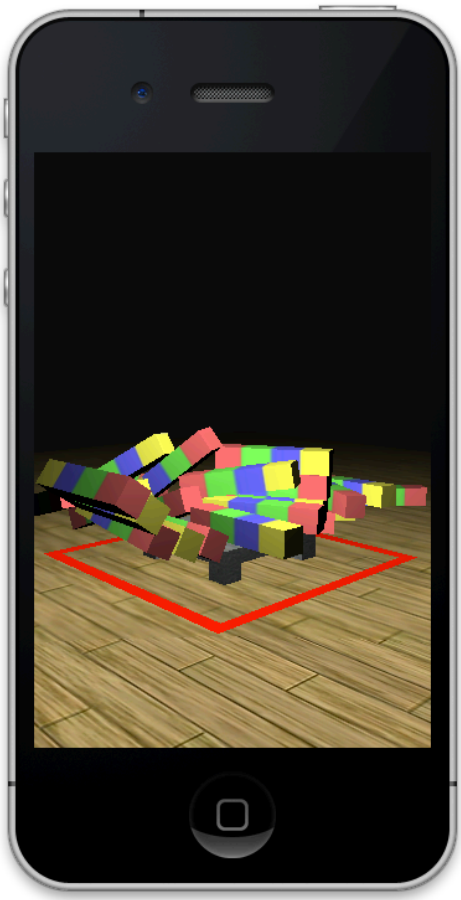
```
        // Is our _btPoint2PointConstraint is different than NULL?
        if( _btPoint2PointConstraint )
        {
             // Remove the constraint from our SIO2physic world.

        sio2GetPhysic()->_btSoftRigidDynamicsWorld-
        >removeConstraint(_btPoint2PointConstraint );

             // Free the memory allocated.
             delete _btPoint2PointConstraint;

             // Reset the pointer to NULL.
             _btPoint2PointConstraint = NULL;
        }
}

void touchesBegan( void *_ptr, SIO2touche *_SIO2touche )
{
     // Temp variable to hold the location that we pick in 3D,
     // as well as the normal vector at this same location.
     vec3 hit_point,
        hit_normal;

     // Create a 3D physic picking ray, if our touche hit a collision object
     // the SIO2object pointer of this object will be returned to us, if nothing
     // is picked the function will simply return NULL.
     selection = sio2PhysicPick( sio2GetPhysic(), // Our global SIO2physic world
information.
                         sio2GetCamera(), // Our global SIO2camera with modelview and
projection matrix updated.
                         sio2GetWindow(),  // Our global SIO2window which contains the
viewport matrix.
                         &_SIO2touche->gles, // Get the OpenGLES color buffer position of
the touche on screen.
                         &hit_point, // If we get a hit, the 3D location will be assigned
to this variable.
                         &hit_normal ); // If we get a hit, the normal vector at the 3D
location will be assigned.

     // Do we have a valid SIO2object pointer? If yes it means that
     // we hit something, now simply create a physic constraint to
     // start dragging our object on screen.
     if( selection )
     {
          // Get the current camera location base on its worldspace matrix.
          btVector3 cam_loc( sio2GetCamera()->_SIO2transform.worldspace.mat[ 3 ][ 0 ],
                             sio2GetCamera()->_SIO2transform.worldspace.mat[ 3 ][ 1
],                                    sio2GetCamera()-
>_SIO2transform.worldspace.mat[ 3 ][ 2 ] );

          // Remove any previously initialized constraint.
          remove_constraint();

          // Set the start position for our drag movement.
          start_pos.setX( hit_point.x );
```

```
          start_pos.setY( hit_point.y );
          start_pos.setZ( hit_point.z );

          // Create a new constraint on the fly based on the btRigidBody information
          // associated with our SIO2 selection object and our start position.
          _btPoint2PointConstraint =  new btPoint2PointConstraint(
                              *selection->_SIO2objectphysic->_btRigidBody,

selection->_SIO2objectphysic->_btRigidBody->getCenterOfMassTransform().inverse() *
                              start_pos );

          // Set that we want to calmp the impulse, this is necessary
          // if we want to achieve smooth drag movements.
          _btPoint2PointConstraint->m_setting.m_impulseClamp = 1.0f;

          // Add the constraint to our physic world.

sio2GetPhysic()->_btSoftRigidDynamicsWorld->addConstraint( _btPoint2PointConstraint );

          // Calculate the vector length (magnitude) between our camera 3D position
          // and our start 3D coordinate.
          pick_dst = ( start_pos - cam_loc ).length();

          // Specify that we don't want the collision object to be deactivated as
          // we want to keep it "alive" as long we are dragging it on screen.

selection->_SIO2objectphysic->_btRigidBody->setActivationState( DISABLE_DEACTIVATION
);
     }
}

void touchesMoved( void *_ptr, SIO2touche *_SIO2touche )
{
     // Do we have a constraint alread created? If yes we must
     // be dragging the block.
     if( _btPoint2PointConstraint )
     {
          // Temp. variable use to calculate the current 3D location of the touche on
screen.
          vec3 curr_point;

          // Get the final camera location in worldspace extracting the XYZ location
from the world matrix.
          btVector3 cam_loc( sio2GetCamera()->_SIO2transform.worldspace.mat[ 3 ][ 0 ],
                       sio2GetCamera()->_SIO2transform.worldspace.mat[ 3 ][ 1 ],
                       sio2GetCamera()->_SIO2transform.worldspace.mat[ 3 ][ 2 ] );

          // Variable use to calculate the current ray position from our
          // camera location to the current 3D point on screen.
          btVector3 ray_to;

          // The direction of our movement (3D ray final position - current 3D point
under the touche)
          btVector3 dir;
```

```
          // Get the 3D location under our 2D touche coordinate.
          sio2UnProject( _SIO2touche->gles.x, // The GLES x position on screen.
                                 _SIO2touche->gles.y,  // The GLES y position on screen.
                                 1.0f, // Since we are picking in 2D, the Z is the far
plane.
                                 sio2GetCamera()->mat_modelview,     // The current
camera modelview matrix.
                                 sio2GetCamera()->mat_projection,    // The current
camera projection matrix.
                                 sio2GetWindow()->mat_viewport,      // The current
window viewport matrix.
                                 &curr_point.x,     // The X position if in 3D of our 2D
coordinate.
                                 &curr_point.y,     // The Y position if in 3D of our 2D
coordinate.
                                 &curr_point.z );   // The Z position if in 3D of our 2D
coordinate.

          // Calculate the direction vector between our current point in 3D and the
camera location.
          sio2Vec3Diff( &curr_point,

( vec3 * )&sio2GetCamera()->_SIO2transform.worldspace.mat[ 3 ],
                                &curr_point );

          // Create the final "ray_to" position in 3D.
          ray_to.setX( cam_loc.x() + curr_point.x );
          ray_to.setY( cam_loc.y() + curr_point.y );
          ray_to.setZ( cam_loc.z() + curr_point.z );

          // Calculate the direction vector of our movement.
          dir = ray_to - cam_loc;

          // Move the pivor point of our constraint so the that block object
          // that we are draggin will move accordingly.
          _btPoint2PointConstraint->setPivotB( cam_loc + ( dir.normalize() * pick_dst
) );
     }
}

void touchesEnded( void  *_ptr, SIO2touche *_SIO2touche )
{
       // When the touche is released, if we have a selection, simply
       // specify that we want the collision object to be deactivated
       // when its velocity become too low.
       if( selection )
       { selection->_SIO2objectphysic->_btRigidBody->setActivationState(
WANTS_DEACTIVATION ); }

       // Remove our 3D constraint (if any)
       remove_constraint();
}
```

We now have a full 3D physic picking code up running. Build and run your application and start

picking and dragging the blocks around in order to test the code.



Game Logic

Since the rules of your game are purely physic based, you will now have to handle different collision callbacks in order to handle our game states. Let's first handle the "game over" state, which is really what determine the end of the game, and allow us to restart the level.

For this you first need to create a collision callback that will notify you when a block touch the physic boundary determined by the limit object that you previously set to a static box collision shape, so it its very easy for you to be able to determine if a block is falling inside the delimitation.

Once again locate the `init` function and paste the following code block at the end just before closing the function bracket:

```
// Enable custom material collision callback for a specific SIO2object.
{
	// Get the limit object pointer from our global SIO2resource.
	SIO2object *_SIO2object = sio2ResourceGetObject( sio2GetResource(), "object/
limit" );
```

```
	// Since we have our collision object created and added to our SIO2physic
	// world, simply enable custom collision callback on it.
	sio2ObjectEnableCollisionObjectCallback( _SIO2object );

	// Assign the global collision callback function. Every contact that happen
	// inside of our physic world will be reported in this function for all
	// collision objects that have been tagged with
sio2ObjectEnableCollisionObjectCallback.
	gContactAddedCallback = MyGameCollisionCallback;
}
```

You have now enable in the code above a global collision callback function, it is now time to actually create that function in code. Simply copy and paste the code below and insert it right before the `init` function declaration.

```
bool MyGameCollisionCallback( btManifoldPoint &_btManifoldPoint,
                              const btCollisionObject *_btCollisionObject0, int p0, int
i0,
                              const btCollisionObject *_btCollisionObject1, int p1, int
i1 )
{
     // As you might have notice from the function declaration, this function will
receive in
     // parameter 2 collision object. For each collision object we will first get back
the
     // SIO2object pointer associated to them by simply casting the collision object
getUserPointer,
     // back to an SIO2object pointer which have automatically been set by the
     // sio2ResourceBindAllPhysicObjects call when we add our SIO2object to our
SIO2physic world.
     SIO2object *_SIO2object0 = ( SIO2object * )( ( btRigidBody * )_btCollisionObject0
)->getUserPointer(),
                *_SIO2object1 = ( SIO2object * )( ( btRigidBody * )
_btCollisionObject1 )->getUserPointer();

     // Check if the name of one of the 2 objects is called limit. If yes
     // it means that one of our block have touch the limit.
     if( ( !sio2StringCmp( _SIO2object0->name, "object/limit" ) ||
           !sio2StringCmp( _SIO2object1->name, "object/limit" ) ) )
     {
          // For now just print game over on the console.
          printf("game over!\n");
     }

     // Continue to process the contact point.
     return 1;
}
```

Lets build and run the game again and drop a block by dragging it to make sure it fall down inside the limit. Then monitor the console for the “game over” print. As you can see as soon as a block touch the limit object, the game over is printed over and over on the command line prompt, which is perfectly normal since the collision callback occur many times per second.

Now lets handle the opposite, and handle the logic if the player actually clear the table without dropping any of the block inside the red limit. For this you are going to implement another type of collision callback function that will act as a “sensor” that will be activated as long as some blocks are actually colliding with the table. To do this add the following line after the `gContactAddedCallback` function callback assignment.

```
// Enable proximity callback when object(s) bounding box overlap.
sio2GetPhysic()->_btCollisionDispatcher->setNearCallback( MyGameNearCallback );
```

Please take note that this time we do not need to enable material collision callback, this function is global to your `SIO2physic` world and will be triggered every time the axis aligned bounding box of an object collide with anohter.

Now Create the `MyGameNearCallback` function and place the following block of code right after the `MyGameCollisionCallback` function so your 2 type of collision callback functions are near each other for a question of convenience.

```
// Declare a counter to count how many current hit we
// got with our table. We are later on going to use that
// counter to determine if the player have successfully
// complete the level.
unsigned int n_table_hit = 0;

void MyGameNearCallback( btBroadphasePair &_btBroadphasePair,
                         btCollisionDispatcher  &_btCollisionDispatcher,
                         const btDispatcherInfo &_btDispatcherInfo )
{
     // Similar as we did in the MyGameCollisionCallback, extract and cast the
btRigidBody
     // user pointer back to an SIO2object pointer.
     btRigidBody
     *_btRigidBody0 = ( btRigidBody * )_btBroadphasePair.m_pProxy0->m_clientObject,
     *_btRigidBody1 = ( btRigidBody * )_btBroadphasePair.m_pProxy1->m_clientObject;

     SIO2object *_SIO2object0 = ( SIO2object * )_btRigidBody0->getUserPointer(),
                      *_SIO2object1 = ( SIO2object * )_btRigidBody1->getUserPointer();

     // Check if one of the two object name that collides against each other
     // is called “object/table”. For a question of convenience we are using the name
     // of the object, however in a real game scenario it is recommended to
     // use the object pointer, which will drastically improve the logic speed
     // especially if we have a large scene with many different condition.
     if( ( !sio2StringCmp( _SIO2object0->name, "object/table" ) ||
           !sio2StringCmp( _SIO2object1->name, "object/table" ) ) )
     {
          // Increase the number of table hit.
          ++n_table_hit;
     }

     // Let Bullet continue to deal with the near callback call for internal
processing.
     _btCollisionDispatcher.defaultNearCallback( _btBroadphasePair,
```

```
                                                    _btCollisionDispatcher,
                                                          _btDispatcherInfo );
}
```

And finally implement the logic code that will determine if the player clear the level. Lets locate the `sio2PhysicRenderFull` function and replace the call block with the following code:

```
// Declare a static float variable to remember the game time.
static float game_time = 0.0f;

// Reset the number of hit we get on the table in the previous pass.
n_table_hit = 0;

// Same as in previous chapter.
sio2PhysicRenderFull( sio2GetPhysic(), sio2GetWindow()->d_time, 2 );

// Increase the game time based on our application delta time.
game_time += sio2GetWindow()->d_time;

// If 2 second elapse, check if we got no hit on the table.
if( game_time >= 2.0f )
{
	// Reset the game time.
	game_time = 0.0f;

	// If we have no hit recorded it means that the table
	// is clean and that the player finish the level. Report
	// in on the console.
	if( !n_table_hit )
	{ printf("level clear!\n"); }
}
```

Now build and run the game and clear the table by moving every block away from the table and outside the delimitation. When you're done, you should see "level clear!" printed on the system console.

As you can see you now have the logic of the game in place, now not much code to plug left before you can make the process loop over and over and in addition, if you had more level make the player go the next level, restart it and so on, like in a real game scenario... but lets first focus on getting one level work for now.

This mark then end of this chapter, as usual you can find the final result of the source code for this chapter under the `SIO2_SDK/tutorials/_MyGame/Chapter3`.

Chapter 4: Game States and Fine Tuning

At this stage you are ready to integrate your game states in order to finalize the game and to be able to run it as long as the player doesn't quit. In addition, in this chapter you will also learn how to add a background music, stream it and play it on loop.

Game State

Locate the following line that have been added in `SIO2sprite *_SIO2sprite = NULL;` and integrate the following block of code right after it:

```
// Definition of our game states.
typedef enum
{
	// The game is running.
	GAME_STATE_PLAY = 0,
	// The game is over and we are waiting for the player input.
	GAME_STATE_GAME_OVER,
	// The player clear the level and we are waiting to go to the next level (if
any).
	GAME_STATE_NEXT_LEVEL

} GAME_STATE;

// The current global game state.
unsigned char game_state = GAME_STATE_PLAY;
```

Now move on to the `MyGameCollisionCallback` function and replace the line `printf("game over!\n");` with the following block of code:

```
// Set the current game state to "game over"
game_state = GAME_STATE_GAME_OVER;

// Toggle our fullscreen SIO2sprite to be visible.
sio2EnableState( &_SIO2sprite->flags, SIO2_SPRITE_VISIBLE );
```

Next lets get back to our `void` init function and locate the following line:

```
sio2EnableState( &_SIO2sprite->flags, SIO2_SPRITE_ENABLED );

// Assign a material to the SIO2sprite.
{
	// Get the game over object.
	SIO2object *_SIO2object = sio2ResourceGetObject( sio2GetResource(), "object/
game_over" );

	// Get the of the first vertexgroup of the game over object
	// and assign it to the SIO2sprite material. The material is
	// also linked to the game over texture and our SIO2sprite
	// is fullscreen. So we are going to use it to display the
	// game over texture depending on our current game state.
	_SIO2sprite->_SIO2material = _SIO2object->_SIO2vertexgroup[ 0 ]->_SIO2material;

	// Change the default blend mode of the material to use the ALPHA value of the
texture.
	_SIO2sprite->_SIO2material->blend = SIO2_MATERIAL_COLOR;
}
```

The next step is to locate inside the `draw_frame` function the block of code between bracket "`{}`" entitled with the following code comment: `// Physic simulation` and add the following condition right before the starting bracket { of the block:

```
if( game_state == GAME_STATE_PLAY )
```

in addition we are also going to add an `else` case that contain the following code:

```
else
{
	// Enter 2D mode
	sio2WindowEnter2D( sio2GetWindow(), -1.0f, 1.0f );
	{
		// Progressively increase the alpha of the diffuse color of our
SIO2material linked
		// to our fullscreen SIO2sprite.
		_SIO2sprite->_SIO2material->diffuse.w += sio2GetWindow()->d_time;

		// Render the SIO2sprite on screen.
		sio2SpriteRender( _SIO2sprite, 1 );

		// Reset the SIO2sprite machine state.
		sio2SpriteReset();
	}
	// Leave 2D mode.
	sio2WindowLeave2D();
}
```

Before going out of the `draw_frame` function replace the following line:

```
printf("level clear!\n");
```

with:

```
// Set the current game state to "next level"
game_state = GAME_STATE_NEXT_LEVEL;

// Toggle the fullscreen SIO2sprite to be visible.
sio2EnableState( &_SIO2sprite->flags, SIO2_SPRITE_VISIBLE );
```

Now inside the `touchesBegan` modify the structure of the function and add an entry condition as in the screenshot below:

```
void touchesBegan( void        *_ptr,
                   SIO2touche *_SIO2touche )
{
    // Are we running the game?
    if( game_state == GAME_STATE_PLAY )
    {...}

    // Wait for the player input to reset the game or load the next level (if any),
    // at this state we must be either in "game over" or "next level" state.
    else
    {
        if( game_state == GAME_STATE_GAME_OVER )
        {
            // The player fail to complete the level, we now need
            // to reset the table.
            reset_game();
        }
        else if( game_state == GAME_STATE_NEXT_LEVEL )
        {
            // If we had other level this would be the location
            // that we would increment the level # and trigger
            // the loading of the next level. In our case we don't
            // have any so we are basically just going to reset
            // the game like in the GAME_STATE_GAME_OVER condition.
            reset_game();
        }
    }
}
```

Here's the code:

```
if( game_state == GAME_STATE_PLAY )
```

And the `else` condition:

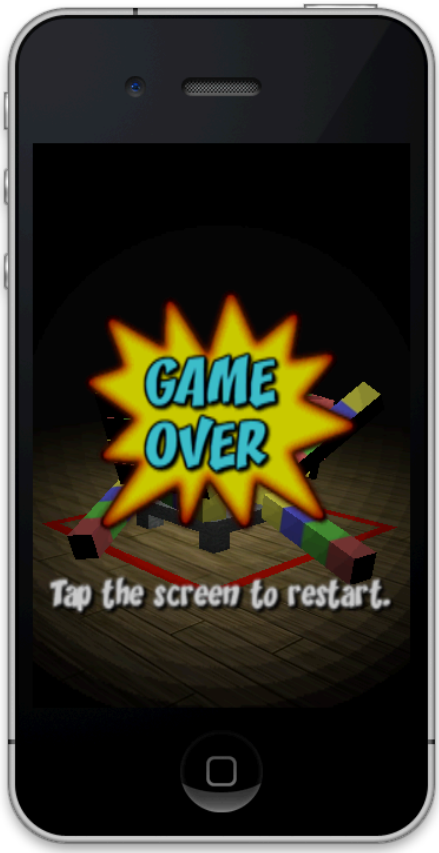
```
else
{
	if( game_state == GAME_STATE_GAME_OVER )
	{
		// The player fail to complete the level, we now need
		// to reset the table.
		reset_game();
	}
	else if( game_state == GAME_STATE_NEXT_LEVEL )
	{
		// If we had other level this would be the location
		// that we would increment the level # and trigger
		// the loading of the next level. In our case we don't
		// have any so we are basically just going to reset
		// the game like in the GAME_STATE_GAME_OVER condition.
		reset_game();
	}
}
```

Time to create a new function called `reset_game` just before the `touchesBegan` declaration:

```
void reset_game( void )
{
```

```
	// Temp. counter.
	unsigned int i = 0;

	// Reset the game state back to play.
	game_state = GAME_STATE_PLAY;

	// Put back the SIO2sprite to invisible.
	sio2DisableState( &_SIO2sprite->flags, SIO2_SPRITE_VISIBLE );

	// Reset its alpha value.
	_SIO2sprite->_SIO2material->diffuse.w = 0.0f;

	// Loop while we got some object inside our global SIO2resource.
	while( i != sio2GetResource()->n_object )
	{
		// Get the SIO2object pointer.
		SIO2object *_SIO2object = ( SIO2object * )sio2GetResource()->_SIO2object[ i
];

	  // Reset the physic object to its original position.
	  sio2PhysicResetObject( sio2GetPhysic(), _SIO2object );
	  // Next object.
	  ++i;
	}
}
```

You are now ready for another test run! So compile and start the game and see how you have successfully pull it off and now have a simply, yet fun, casual game running on your device. Try to get game over by dropping a block inside the limit and try to clear the table to "`goto`" the next level (even if you only have one but). Your game should now look like in the figure below:

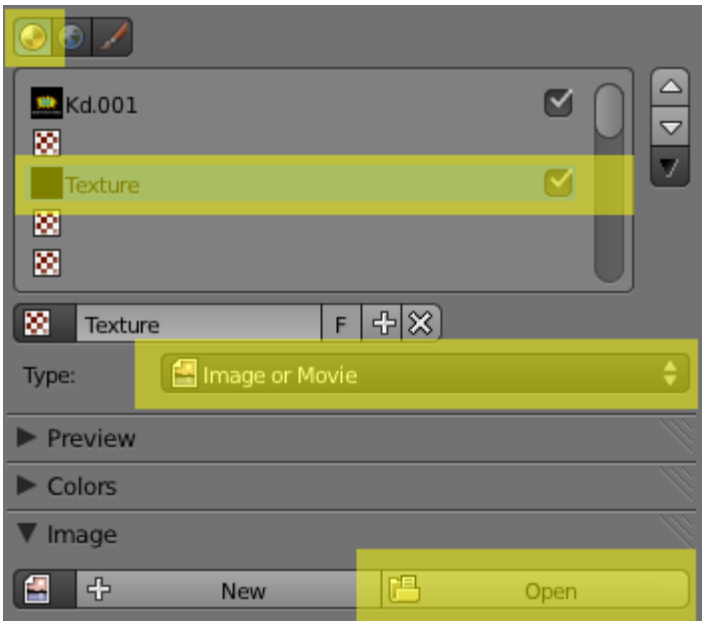


Background Music

Time to add a bit of music to the game, for this re-open your Blender file and select the game_over object by right clicking on it.

Then click the Texture button  in order to access the textures properties. In order for SIO2 to be able to handle 3d (positional) or ambient (static) sound source you have link them like if it was a texture, since Blender is not offering any other way to link sounds to a specific material, you have to use the texture slot, starting from the 3rd one up until the 10th (max. 8 sound buffer per material is supported by default in SIO2).

So first select the 3rd texture channel, then click “Add New”, select the Texture Type to be “Image or Movie” and click the “Open” button:



You should then see the Blender file dialog, but before you can select the OGG file that you are going to use for sound you have to toggle OFF the file filtering in Blender as a `.ogg` is not a format that

Blender can recognize, to do that simply click the  button at the top of the file dialog. Then browse to the `SIO2_SDK/tutorials/_MyGame/Chapter4/` directory and select the `music.ogg` sound file. By default, every sound buffer added will be considered as an ambient sound, streamed and ready to play and to loop over and over. For more details about the other options that you can use with sound file please consult the exporter manual.

Now export your scene again to overwrite the current .sio2 the same way as you did in the “Physic and Instancing” section where you toggle to update the .sio2 and export the vertex normals.

Now back to `template.cpp`, in order to start playing the background music, you have to add a few lines of code. First jump to the `init` function and add the following instruction after the `sio2Init` function:

```
// Initialize OpenAL
sio2InitAL();
```

Next you have to tell SIO2 to link the sound buffer to the material, this is done using the following line:

```
// Link all the sound buffers to their respective materials.
sio2ResourceBindAllSoundBuffers( sio2GetResource() );
```

and place it right after the `sio2ResourceBindAllImages` function call.

The next step is to create the sound source, the approach SIO2 is using allow you to have one sound buffer in memory and then share it among multiple sound sources. However for this tutorial you only have one so we won’t be using this technique, but in any case you still have to bind all

the sound buffers and create a new sound source. To do this plug the following code right after the `sio2ResourceGenId` function call:

```
// Create a unique sound source for each sound buffer attached to the SIO2object
material.
sio2ResourceBindAllSounds( sio2GetResource() );
```

And now the final touch to make the sound be streamed from memory and playback using OpenAL you need to tell to the `sio2ResourceRender` function to actually stream the sound buffer and to deal with the quad buffering operations in real time, to do this simply replace the `sio2ResourceRender` function call with the following:

```
sio2ResourceRender( sio2GetResource(),
                    NULL,
                    NULL,
                    SIO2_RENDER_SOLID_OBJECT    |
                    SIO2_RENDER_CLIPPED_OBJECT |
                    SIO2_UPDATE_SOUND_STREAM  | // Update the sound stream.
                    SIO2_RENDER_LAMP );
```

In order to make sure that you clean the OpenAL state when the game close add the following line before the `sio2Shutdown` function inside the `close_app` function callback:

```
// Shutdown OpenAL
sio2ShutdownAL();
```

Now re-compile the game for the last time and run and enjoy!

Conclusion

In this getting started tutorial you have learn a lot! Now the next step, in order to get the most out of SIO2 is to start digging inside the different tutorials available in the `SIO2_SDK/tutorials/`. Read carefully the source code documentation for each of them, and pay close attention to every properties set inside their respective `.blend` file.

In addition, do not hesitate to consult http://api.sio2interactive.com for more information about the SIO2 APIs as well Exporter Manual for your favorite 3d software, in order to learn what is supported by the exporter and what is not.

If you have a questions regarding the code or the assets, or if you simply want more information about SIO2 v2.x or the SDK feel free to post your quesion(s) online at http://forum.sio2interactive.com in the appropriate forum thread.

SIO2 is quite a large 2d/3d engine and the learning curve might look steep at first, but with a bit of time and patience you will realize how fast, flexible, scalable and powerful it is! With SIO2 sky is the limit, there's no restrictions!

Hope you enjoy this getting started, enjoy your SIO2 Trial!